Treball defi de grau

GRAU EN ENGINYERIA ELECTRÒNICA INDUSTRIAL I AUTOMÀTICA

UNIVERSITAT POLITÈCNICA DE CATALUNYA

ESCOLA POLITÈCNICA SUPERIOR D'ENGINYERIA DE MANRESA

PIANO DIGITAL

MIDI a S/PDIF

GUILLEM IBÁÑEZ MASANÉS

Tutor: David Soler Jiménez

Curs acadèmic: 2018 - 2019

RESUM

L'objectiu d'aquest projecte és crear un piano funcional fent ús d'una FPGA i un teclat MIDI.

Per norma general un teclat MIDI no emet música per si sol; és dependent d'un ordinador i d'un programari específic. En aquest projecte, l'FPGA substituirà l'ordinador, i el programari específic es crearà mitjançant el VHDL, un llenguatge de baix nivell.

Podem dividir el funcionament de tot piano digital en dues parts essencials: la creació i la reproducció de la música.

Assegurarem aquest funcionament gràcies al protocol MIDI, per produir la música, i l'estàndard S/PDIF, per a fer-la sonar.

En les pàgines següents descobrirem què són el MIDI i l'S/PDIF, i veurem com els dos treballen i es coordinen amb el cúmul d'entitats VHDL que faran que aquest piano digital funcioni correctament.

SUMMARY

The aim of this project is to create a functional piano using a MIDI keyboard and a FPGA.

MIDI keyboards do not usually emmit music on their own; they are dependent from a computer and specific programs. In this project, the FPGA will take the place of the computer, and all required programs will be written in VHDL, a low-level programming language.

We can divide the functionality of a digital piano in two main steps: the creation and the reproduction of music.

These functions will be assured thanks to MIDI protocol, which will be in charge of generating the music, and the S/PDIF standard, that will let the music sound.

In the following pages we are going to discover what MIDI and S/PDIF protocols consist of, and we will see how both of them work together and coordinate themselves with a series of VHDL entities, ensuring that this piano works correctly.

ÍNDEX

1. GLOSSARI

1. **DAC (Digital to Analog Converter)** → Dispositiu que converteix senyals digitals a analògiques.

2. **TTL (Transistor-Transitor logic)** → Tipus de tecnologia de fabricació de circuits electrònics.

3. **USB (Universal Serial Bus)** → Estàndard de connectors, cables i protocols de comunicació que serveixen per alimentar i/o comunicar diferents dispositius electrònics.

4. **MIDI (Musical Instrument Digital Interface)** → Protocol de comunicació entre diferents instruments i dispositius electrònics dedicat a la creació musical.

5. **FPGA (Field Programmable Gate Array)** → Dispositiu electrònic que conté blocs lògics programables. Sovint disposa de complements com interruptors, botons, LEDs, etc. configurables gràcies als blocs lògics.

6. **S/PDIF (Sony Philips Digital Interface Format)** → Protocol de transmissió en sèrie de senyals d'àudio.

7. **UART (Universal Asynchronous Receiver-Transmitter)** → Entre dispositius, circuit de comunicació asíncron, bidireccional i en sèrie.

8. **IC (Integrated Circuit)** → Dispositiu de petites dimensions i constituït d'un material semiconductor en el que es fabriquen circuits electrònics. Per norma general, un IC ve encapsulat en ceràmica o plàstic.

9. **VHDL (VHSIC Hardware Description Language)** → Llenguatge de programació a nivell de hardware. Especialment útil en la programació de blocs lògics de l'FPGA.

10. **VHSIC (Very High Speed Integrated Circuit)** → Programa militar d'EUA de recerca i desenvolupament d'IC de gran velocitat d'operació.

2. INTRODUCCIÓ

MIDI és el protocol de comunicació entre instruments musicals electrònics més comú arreu del món. Pocs fabricants han decidit desmarcar-se'n, doncs és un estàndard tant complet com eficaç que des de l'any 1982 ha dominat la indústria musical. Tant el podem trobar a casa d'un teclista aficionat de Manresa com en les taules de so dels mateixos Red Hot Chili Peppers. Tots hem participat del MIDI en algun moment de les nostres vides i no n'hem sigut conscients.

Si bé el MIDI ens permet fer música, de poc ens serveix si no la podem escoltar. L'estàndard S/PDIF és més avorrit, però això no el fa menys indispensable. Dins dels protocols d'àudio digital, l'S/PDIF va ser un dels més exitosos degut a la seva simplicitat i eficiència, abans que l'HDMI el destronés.

Aquest projecte tracta d'unir ambdós estàndards en els blocs lògics d'una FPGA, que actuarà de nexe entre un teclat i un altaveu, que per si sols no sonen ni s'entenen.

3. QUÈ ÉS MIDI?

A les acaballes del segle XIX, l'inventor nord-americà Thaddeus Cahill va desenvolupar l'anomenat Telharmonium, un teclat elèctric de grans dimensions capaç de produir i mesclar sons diversos. Aquest aparell, construït sobre els passos del metge i físic alemany Herman von Helmholtz, que tres dècades abans havia ideat una màquina capaç de produir sons nets mitjançant generadors de corrent alterna, és considerat el primer sintetitzador de la història.

Tot i així, com tants altres invents de l'època, el Telharmonium va necessitar molts anys per polir-se. Era complicat de tocar, els seus problemes tècnics, incomptables, i des de la seva primera aparició el 1897 fins la presentació de la seva versió darrera l'any 1918, poques persones van interessar-se per l'instrument.

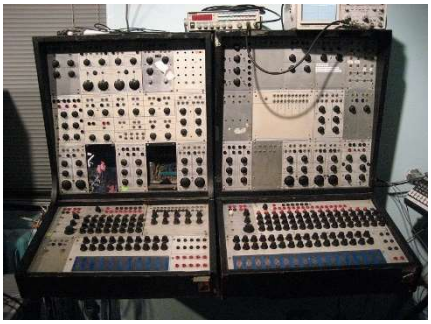
Imatge 1. Thaddeus Cahill tocant l'immens Telharmonium (1906).

El fracàs del Telharmonium, sumat a les mancances tecnològiques de principis del segle XX, van constituir una barrera per l'evolució dels teclats i sintetitzadors elèctrics; els invents de Von Helmholtz i Cahill quedarien en l'oblit i ningú gosaria desempolsar les seves empreses fins passades quatre dècades.

L'aparició de l'electrònica i dels primers instruments elèctrics comercials a principis dels seixanta, van donar a conèixer Harald Bode i el seu Melochord, el primer sintetitzador electrònic controlat per voltatge. Malgrat que el Melochord es tractés encara d'un sintetitzador força primitiu en quant a complexitat i versatilitat, impulsaria sense saber-ho una revolució en la indústria musical.

Seguint els passos de Bode, dels nord-americans Robert Moog i Donald Buchla traurien al mercat els seus propis sintetitzadors entre el 1964 i 1966, que molt originalment anomenarien Moog i Buchla. La creixent popularitat d'aquests instruments donaria peu

al naixement del sector dels sintetitzadors electrònics, i sorgirien nombroses empreses que dedicarien grans esforços a la millora tecnològica dels seus productes.



Imatge 2. Buchla Series 100, primer sintetitzador comercialitzat per D. Buchla (1966).

Any rere any nous sintetitzadors arribarien al mercat, cada un d'ells amb més funcionalitats i característiques novedoses. Va ser aquesta creixent complexitat del sector que va permetre a principis dels vuitanta l'aparició dels primers *samplers* comercials i seqüenciadors. Aparells que no estaven pensats per funcionar en solitari, sinó per coordinar diferents instruments.

De cop i volta, els fabricants van llençar-se a desenvolupar els seus propis protocols de comunicació que permetrien coordinar diferents instruments electrònics sonant al mateix temps. Però el pas del temps demostraria que això era insostenible, doncs només aparells de la mateixa marca podien comunicar-se entre si reduint el ventall d'instruments als que un músic podia optar si volia que sonessin alhora.

Aquesta mancança va portar a diferents fabricants nord-americans a plantejar-se la creació d'un protocol únic que permetés que, independentment de la casa, un instrument pogués entendre's amb qualsevol altre. Aquesta idea es va posar sobre la taula als Estats Units l'any 1981 i als pocs mesos va néixer l'estàndard USI (Universal Synthesizer Interface). L'USI permetia una comunicació en sèrie de 19.200 *bps* amb nivells lògics TTL de 0 a 5 volts.

L'èxit d'aquesta iniciativa va deixar-se veure un any després a la NAMM (National Association of Music Merchants), on diferents empreses nord-americanes van decidir portar-lo més enllà de les seves fronteres. Fabricants japonesos van bolcar en el projecte els seus coneixements resultant en millores significatives. Ara, l'estàndard USI permetia una comunicació en sèrie de 31.250 *bps*, i amb presència creixent a tot el globus va rebatejar-se com a MIDI (Musical Instrument Digital Interface).

El primer instrument en fer servir aquest protocol va ser el Prophet600 de Sequencial Circuits. MIDI es va colar al mercat musical a través seu, i al cap d'uns mesos ja l'havia conquerit.

Imatge 3. Prophet600, primer sintetitzador comercial en emprar el MIDI (1982).

L'aparició del pop dels vuitanta va propiciar un èxit en vendes dels sintetitzadors electrònics, que van esdevenir indispensables en qualsevol grup i cançó. Si MIDI va provocar el *boom* del pop o viceversa, és un tema de debat.

Sorprenentment, la simplicitat i eficàcia del MIDI l'han mantingut aliè a les revolucions tecnològiques subsegüents, i a dia d'avui, més de tres dècades després del seu naixement i havent sofert minses actualitzacions, segueix sent el protocol de comunicació musical més estès arreu del món.

Per crear i reproduir el so, el piano digital del present projecte es basa en el protocol MIDI per poder desglossar la informació del teclat i processar-la. En les següents pàgines s'explicarà en què consisteix aquest protocol; com s'estructura, quina informació duu i per a què la fem servir. Finalment, connectarem el teclat, tocarem algunes tecles i analitzarem detalladament la informació entrant per entendre com es comunica un dispositiu MIDI.

3.1 Missatges MIDI

Els missatges MIDI són les frases fonamentals que permeten comunicar els dispositius MIDI entre ells amb l'objectiu comú de descriure el so que haurà d'emetre's pels altaveus.

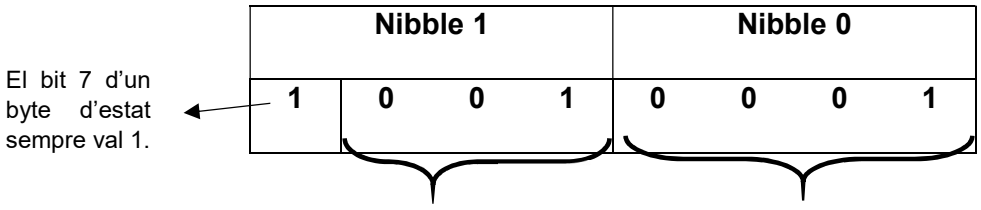
Es componen d'un seguit de bytes (8 bits) que es transmeten en sèrie, bit per bit, amb un ordre establert. Al davant porten un bit start (0), i al darrere, un bit stop (1). La freqüència de transmissió és, per norma general, de 31.250 bps.

Existeixen dos grans grups de bytes en els missatges MIDI: els bytes d'estat i els bytes de dades.

3.1.1 Bytes d'estat

Els bytes d'estat (*status byte*) ens indiquen les instruccions i el canal on s'efectuaran. Tanmateix, tot byte d'estat necessita un o més bytes de dades per complementar les seves ordres.

Dins d'un missatge MIDI, el byte d'estat sempre va al davant seguit pels bytes de dades. La seva estructura (excloent els bits Start i Stop mencionats al principi) és la següent:



Bits 6, 5 i 4 corresponen a la instrucció a executar. Tres bits significa que podem escollir entre 8 instruccions diferents. Les veurem a continuació.

Els bits 3, 2,1 i 0 del byte d'estat corresponen al canal on s'aplicarà la instrucció donada als 3 bits menys significatius del Nibble 1. Quatre bits dedicats a l'elecció del canal ens donen un rang de 16 canals amb els que treballar.

3.1.2 Bytes de dades

Els bytes de dades (*data bytes*) són la informació que, definida pels bytes d'estat precedents, permeten manipular les dades entrants del dispositiu i, consegüentment, el so final. Es diferencien dels bytes d'estat no només perquè van al seu darrere sinó perquè el MSB (bit més significatiu) de cada un d'ells sempre val 0.

Per si sols, els bytes de dades no signifiquen res, doncs la forma en que el seqüenciador els llegeixi dependrà totalment del byte d'estat que els encapçali, de manera que els seus formats poden resultar confusos per no dir que inintel·ligibles, si es miren independentment. Per això, a continuació, veurem com es combinen els bytes d'estat i els bytes de dades per donar forma als diferents missatges MIDI.

3.2 Tipus de missatges MIDI

Els missatges MIDI són comptats i comuns en la gran majoria de dispositius MIDI, doncs aquesta és la clau perquè entre ells s'entenguin i puguin treballar junts en una mateixa cançó. Cert és que alguns dispositius molt simples no contemplen certs missatges dels que disposen algunes màquines d'alta gamma i que existeixen instruccions específiques de cada fabricant per als instruments de la seva casa. Tanmateix, els missatges essencials són inalterables i comuns en tots els dispositius.

L'estructura dels missatges MIDI varia completament en funció del byte d'estat (que determina la seva utilitat i focus d'acció). Intentarem comprendre aquests missatges, definint-los i analitzant la seva composició.

Però abans d'endinsar-nos en l'explicació de cadascun d'aquests missatges ens aturarem un moment amb la taula següent, que ens servirà de guia en els següents apartats.

MISSATGE	**Byte d'estat**		**Nº Bytes de dades**	**Byte de dades 1**	**Byte de dades 2**
	MS Nibble	**LS Nibble**			
Note Off	0x8	Canal (0→15)	2	Número de nota	Velocitat
Note On	0x9	Canal (0→15)	2	Número de nota	Velocitat
Polyphonic Pressure	0xA	Canal (0→15)	2	Número de nota	Pressió

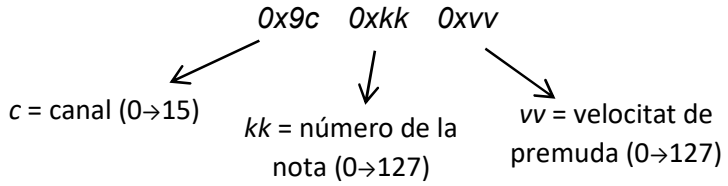
Control Change	0xB	Canal (0→15)	2	Número del controlador	Valor
Program Change	0xC	Canal (0→15)	1	Número de programa	N/A
Channel Pressure	0xD	Canal (0→15)	1	Pressió	N/A
Pitch Bend	0xE	Canal (0→15)	2	LSB Curvatura	MSB Curvatura
System	0xF	Variable	Variable	Variable	Variable

Taula 1. Tipus de missatges MIDI.

Note On (0x9)

Aquesta comanda s'activa quan premem una tecla (en el cas d'un teclat). Els seus bytes de dades indiquen el número de nota i la velocitat amb la que aquesta tecla s'ha premut determinant, així, la tonalitat i el volum.

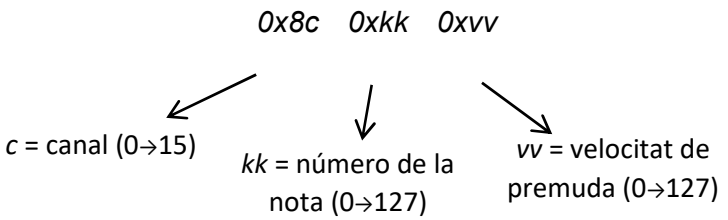
La seva estructura (en bytes representats en hexadecimal) és la següent:



Note Off (0x8)

La comanda Note Off és l'antítesi de Note On. S'activa una vegada alliberem la tecla per eliminar el so. De la mateixa manera que Note On, disposa de dos bytes de dades. El primer ens indica el número de la nota a silenciar i el segon la velocitat.

La seva estructura, doncs, és similar a Note On:

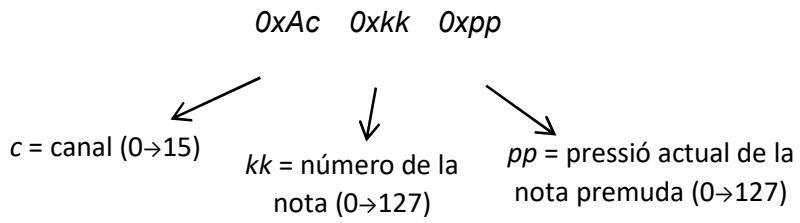


Alguns dispositius MIDI no incorporen el Note Off en essència, com el Keystation Mini 32. En el seu lloc, el Note Off és substituït per un segon Note On amb el byte de velocitat nul, sobreescrivint la nota amb volum zero.

Polyphonic Pressure (0xA)

És una instrucció poc comuna que dota cada nota de la capacitat d'enviar informació independent sobre la seva velocitat de premuda. Té dos bytes de dades corresponents el primer al número de nota i a la velocitat.

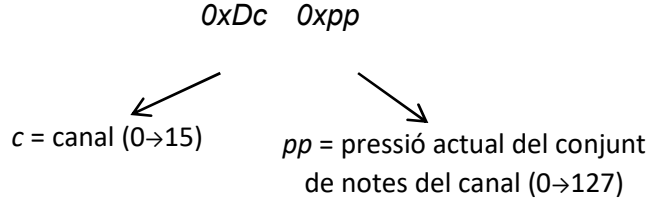
S'estructura així:



Channel Pressure (0xD)

El Channel Pressure permet a un canal enviar informació general sobre la pressió actual de les tecles sonants amb un valor únic de velocitat que permet modificar-se prement més tecles, canviant el seu volum sense interrompre's. Per fer-ho, necessita un byte d'estat, indicant la instrucció de Channel Pressure, i un sol bytes de dades corresponent a la pressió de les tecles del canal.

La seva forma és la següent:



Pitch Bend (0xE)

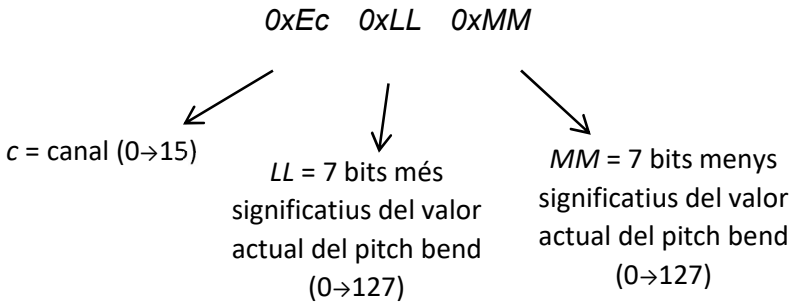
El Pitch Bend és un efecte sonor que “doblega” progressivament la nota cap a greu o cap a agut mitjançant, generalment, una rodeta que trobem a gairebé tots els teclats.

El byte status del Pitch Bend ve seguit de dos bytes de dades (7 bits usables per cada un) que determinen el valor d'aquesta curvatura del so. En total, el Pitch Bend, doncs, disposa d'un rang de 16.383 (2^{14}) valors per a definir-se. El valor per defecte (sense Pitch Bend) es troba a la meitat d’aquest rang, és a dir, al valor 8.192.

En el cas del Keystation Mini 32, el controlador del Pitch Bend es comprèn de 2 botons (un a l'esquerra del botó MOD per corbar la nota cap a greu, i un a la dreta per pujar-la).

Imatge 4. Controlador del Pitch Bend del Keystation Mini 32 (en groc).

La seva estructura és aquesta:

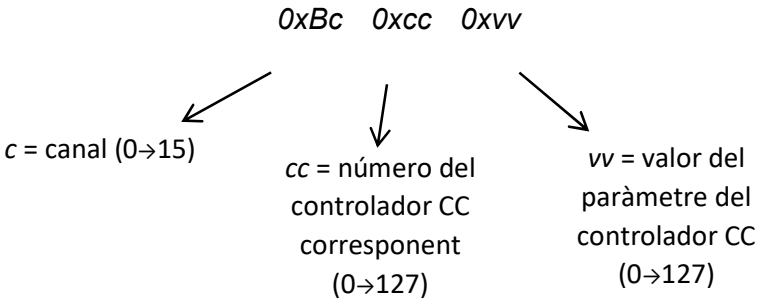


Control Change (0xB)

El Control Change (canvi de control) gestiona els controls d’expressió, altrament coneguts com a Continuous Controllers o Canvis de Control (CC). Amb aquest nom es coneixen els *sliders*, rodetes, botons, etc. que ajuden a modificar el so (excloent el Pitch Bend, que ja té una instrucció pròpia degut a la quantitat de bits que necessita per desenvolupar-se correctament).

El Control Change disposa de dos bytes de dades; el primer és el número de CC, i el segon és el valor del CC escollit. Cada byte es comprèn de 7 bits útils, i per tant tenim 128 CC diferents (que veurem als Annexos) i un rang de 128 valors per a cada un.

Vegem la seva estructura:



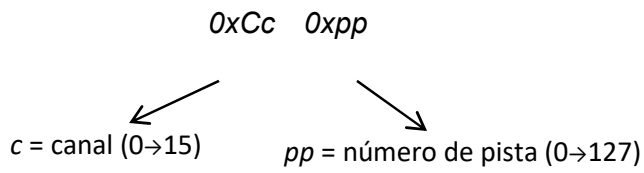
El teclat Keystation Mini 32 disposa de 3 controladors que, per defecte, s'encarreguen del volum i el *sustain*.

Imatge 5. Controladors del volum (a dalt) i sustain (a baix) del Keystation Mini 32.

Program Change (0xC)

El Program Change ens permet canviar el so que s'està emprant per un altre. Entenem per so com a l'arxiu d'àudio fonamental que després pot modular-se per assolir les diferents notes, i per tant, és la veu de l'instrument. A diferència de les altres instruccions vistes fins ara, el Program Change sols necessita un byte de dades (de 7 bits útils) que és el "patch number", o número de pista, emmagatzemat. Per norma general, disposem de 128 sons o pistes diferents en un dispositiu MIDI normal.

El format del Program Change és el següent:

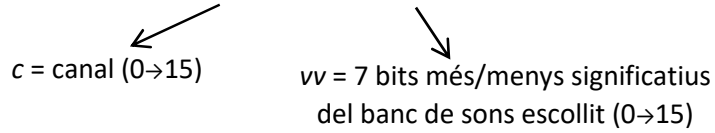


Per jugar amb més de 128 sons tenim els anomenats bancs; espais extres de memòria accessibles mitjançant comandes específiques a les que anomenem "bank select".

En el cas del Keystation Mini 32 disposem de més de 128 pistes diferents, que podem seleccionar canviant de banc a través de les comandes de Control Change següents:

0xBc *0x00* *0xvv* Bank MSB

0xBc *0x20* *0xvv* Bank LSB



System Messages (0xF)

Fem servir els System Messages per executar qualsevol instrucció que no hem pogut incloure en les instruccions anteriors. Es divideixen en System Realtime Messages, System Common Messages (o Non-Realtime Messages) i System Exclusive Messages.

Podem definir els System Realtime Messages com a els System Messages que s'utilitzen per sincronitzar els seqüenciadors i que no necessiten bytes de dades que els complementin.

Els System Realtime Messages típics són els següents:

- *Timing Clock*
  Estableixen el tempo de treball del seqüenciador, que ve predeterminat a 24 temps per negra.
- *Start*
  Reinicia la cançó.
- *Continue*
  Reprèn la reproducció de la cançó des del punt on s'havia pausat.
- *Stop*
  Atura la reproducció de la cançó.
- *Active Sense*
  És una comanda opcional que, en cas de que estigui activa, envia un missatge cada 300 mil·lisegons amb l'únic fi de notificar que el dispositiu està connectat. Si passat aquest temps no s'ha rebut el missatge, es para el seqüenciador i totes les notes deixen de sonar.
- *System Reset*
  Atura automàticament totes les notes actives. Serveix principalment en el cas que un Note Off sigui ignorat, doncs el so pot impedir el correcte funcionament del dispositiu o fins i tot arribar a augmentar en amplitud de forma descontrolada.

Els System Common Messages són les instruccions que sí que requereixen bytes de dades i que no són tan essencials com els System Realtime Messages. Aquestes instruccions són les següents:

- *Time Code Quarter Frame*
  Estableix un temps absolut de sincronització. El byte de dades que prossegueix indica aquest temps.
- *Song Position*
  Permet saltar a un punt concret de la cançó que s'estigui reproduint. Els dos bytes de dades que succeeixen al byte d'estat determinen la posició exacta on s'ha de saltar.
- *Song Select*
  Reprodueix una nova cançó. El byte de dades que succeeix a la instrucció determina la cançó seleccionada.
- *Tune Request*
  Afina al dispositiu que ocupa el canal seleccionat. No necessita bytes de dades.

Els System Exclusive Messages, altrament anomenats SysEx, són missatges que serveixen per a enviar informació específica a un dispositiu o a varis d'interconnectats. Aquesta informació es comprèn de varis bits de dades destinats a millorar certes característiques de so sortint del seqüenciador. No tots els dispositius MIDI disposen de la capacitat de processar missatges SysEx; només els que incloguin funcionalitats pròpies de fabricant pensades per activar-se de forma exclusiva per aquests missatges.

Els SysEx tenen, no un, sinó dos bytes d'estat: un al davant, anomenat Start Of Exclusive (SOX) amb valor 11110000, i un al darrere, anomenat End Of Exclusive (SOX) i amb valor 11110111. Enmig, va el missatge pròpiament dit.

El format dels System Messages és variable. Per entendre'l millor ens ajudarem de la taula següent:

Missatge	**Tipus**	**Byte d'estat**	**Nº bytes de dades**
SOX (Start Of Exclusive)	SysEx	0xF0	Variable
Time Code Quarter Frame	Non-Realtime	0xF1	1

Missatge	Tipus	Byte d'estat	Nº bytes de dades
Song Position	Non-Realtime	0xF2	2
Song Select	Non-Realtime	0xF3	1
Undefined	Non-Realtime	0xF4	0
Undefined	Non-Realtime	0xF5	0
Tune Request	Non-Realtime	0xF6	0
EOX (End Of Exclusive)	SysEx	0xF7	0
Timing Clock	Realtime	0xF8	0
Undefined	Realtime	0xF9	0
Start	Realtime	0XFA	0
Continue	Realtime	0xFB	0
Stop	Realtime	0xFC	0
Undefined	Realtime	0xFD	0
Active Sense	Realtime	0xFE	0
System Reset	Realtime	0xFF	0

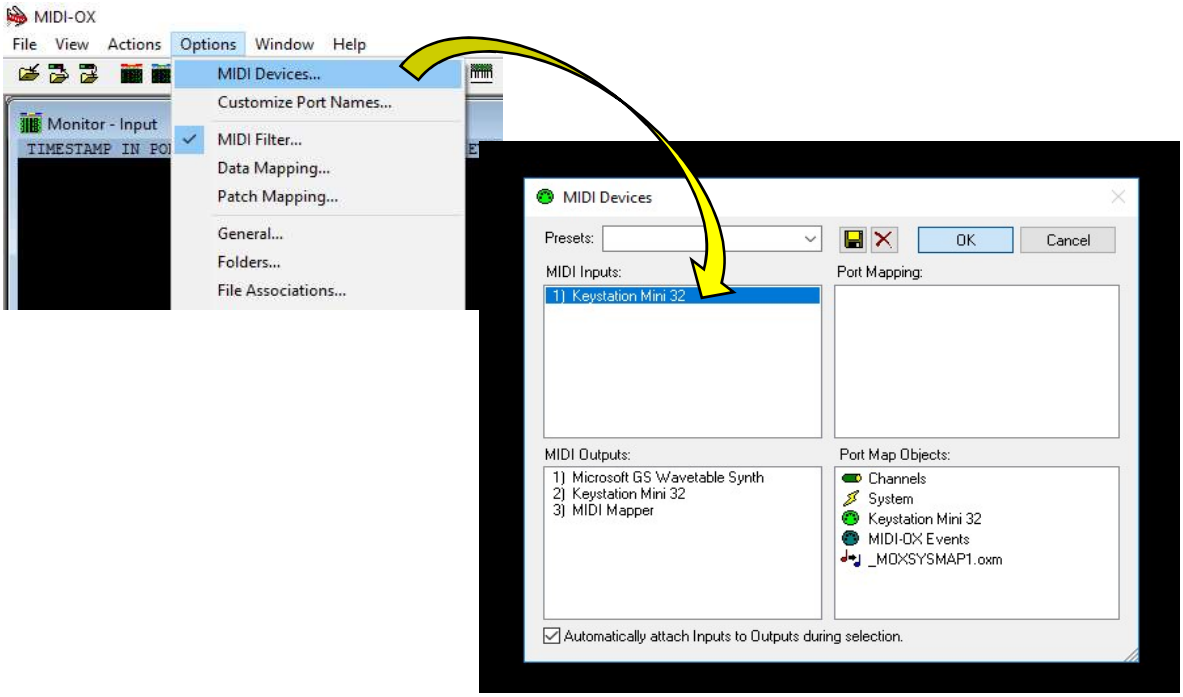
Taula 2. System Messages de MIDI.

3.3 Desglossament de la senyal MIDI

Ara que tenim entesa la teoria, veurem com el dispositiu MIDI es comunica amb l'ordinador. Per fer-ho, existeixen alguns programes específics, com el Pocket MIDI o el MIDI-OX, que, seleccionant el port USB pel que entren les dades MIDI, llegeixen la informació entrant, la transformen per fer-la intel·ligible i la mostren per pantalla. Nosaltres farem servir el MIDI-OX, doncs la interfície d'ambdós és igual de senzilla i per la feina que els requerim qualsevol dels dos programes ens és vàlid. No obstant, el MIDI-OX ens desglossa millor la informació entrant i la fa més païble.

Imatge 6. Icona de MIDI-OX.

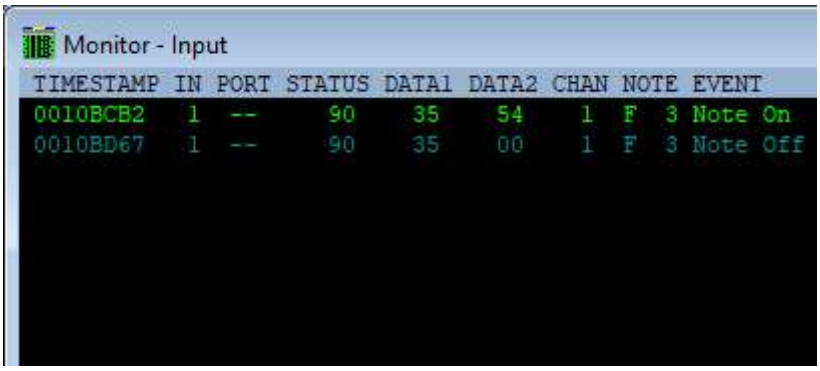
Una vegada connectat el teclat Keystation Mini 32 a l'ordinador mitjançant un dels ports USB, engeguem l'aplicació del MIDI-OX i seleccionem el nostre dispositiu.



Un cop cliquem OK ens desapareix la finestra "MIDI Devices" i retornem a la pantalla en negre que domina la interfície de l'aplicació. Ara, quan premem qualsevol tecla, la informació entrarà a l'ordinador i MIDI-OX la transformarà perquè la puguem llegir i interpretar.

Començarem provant una **tecla a l'atzar**, que premerem i alliberarem seguidament. Si tot funciona bé, hauríem de veure com s'activa un Note On al canal 0 i amb una velocitat

determinada, per després activar-se un Note Off (en el Keystation Mini 32 equivalent a un altre Note On amb velocitat 0) al mateix canal.



Com veiem, MIDI-OX no només ens desglossa la informació entrant sinó que a més ens la classifica.

A la primera columna (TIMESTAMP) ens mostra, en hexadecimal, el temps transcorregut des que s'ha engegat l'aplicació fins que s'ha enviat el missatge MIDI.

El bit d'estat (STATUS) ens indica la comanda i el canal al que aplica, com hem vist anteriorment.

Els bits de dades (DATA1 i DATA2) ens donen informació sobre la comanda. Com que aquesta es tracta de Note On i Note Off, sabem que DATA1 fa referència a la nota tocada, que podem comprovar que és la F3 (fa_2) a través de la columna NOTE, i DATA2 mostra la velocitat de premuda de la tecla, recordem, definint la potència de la nota.

La columna CHAN ens diu el canal al que s'està aplicant la comanda i EVENT ens ajuda a entendre de quina comanda es tracta sense haver de consultar enlloc. En aquestes dues últimes columnes podem observar alguns detalls interessants:

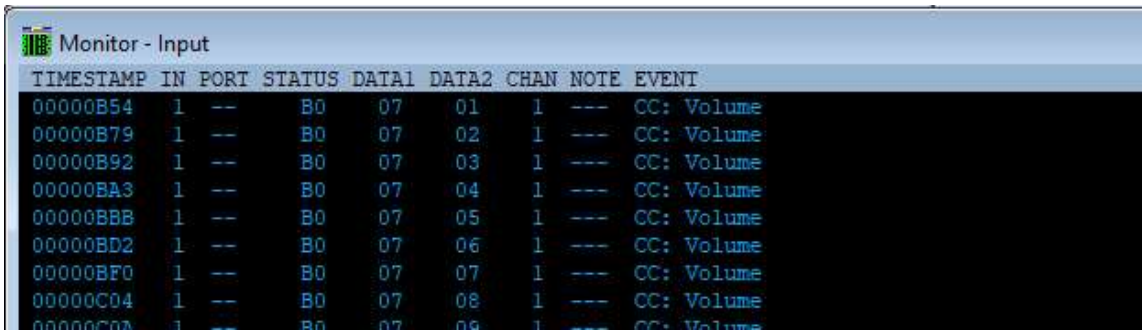
a) El nibble menys significatiu del byte d'estat en els dos casos és 0, però la columna del canal (CHAN) ens indica que les comandes s'han aplicat al canal 1.
b) El nibble més significatiu de la segona comanda (Note Off) és exactament igual que el Note On.

No, l'aplicació no s'ha tornat boja i el teclat fa la seva feina correctament. El que passa és que alguns dispositius MIDI funcionen lleugerament diferent que alguns altres, encara que això no suposi un impediment a l'hora de treballar plegats, i en altres ocasions som les persones que entenem de maneres diferents conceptes que són els mateixos. Expliquem-nos:

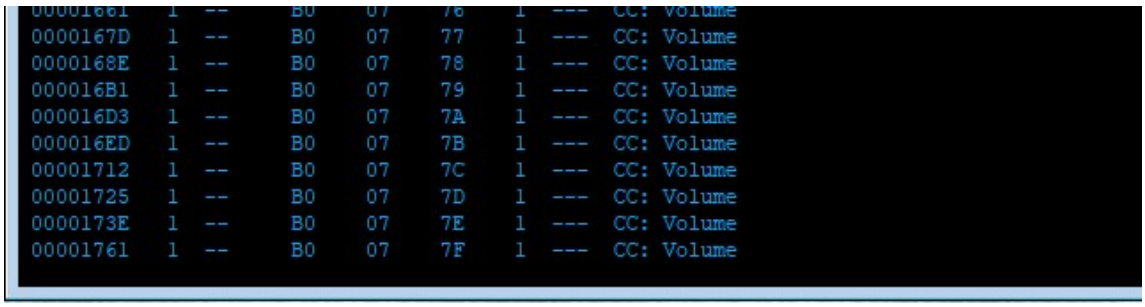
El primer cas el podem atribuir a la interpretació que fa MIDI-OX de la lectura del canal. Degut a la confusió que pot provocar saber que hi ha 16 canals diferents però que a l'hora de comptar-los vagin del 0 al 15, molts fabricants aposten per anomenar-los de l'1 al 16. Òbviament, a l'hora d'establir el protocol MIDI, no han afegit un bit extra per simplement poder establir un canal 16; l'únic que canvia, per dir-ho d'alguna manera, és el nom que li donen als canals en els seus manuals.

El segon cas és degut a que alguns dispositius, per tal de simplificar els seus missatges MIDI i fer més efectiu l'ús del byte d'estat, substitueixen el Note Off per un altre Note On amb velocitat 0, sobreescrivint la nota tocada per una d'igual però amb volum 0, inhibint-la. A efectes pràctics, el resultat és el mateix.

Podem provar altres comandes, com el **volum** (girant el controlador de rodeta). Al principi la rodeta està posicionada al seu màxim a l'esquerra (byte DATA2 = 00), equivalent a un nivell de volum nul. Per comprovar el seu rang, la girarem fins a l'extrem dret.

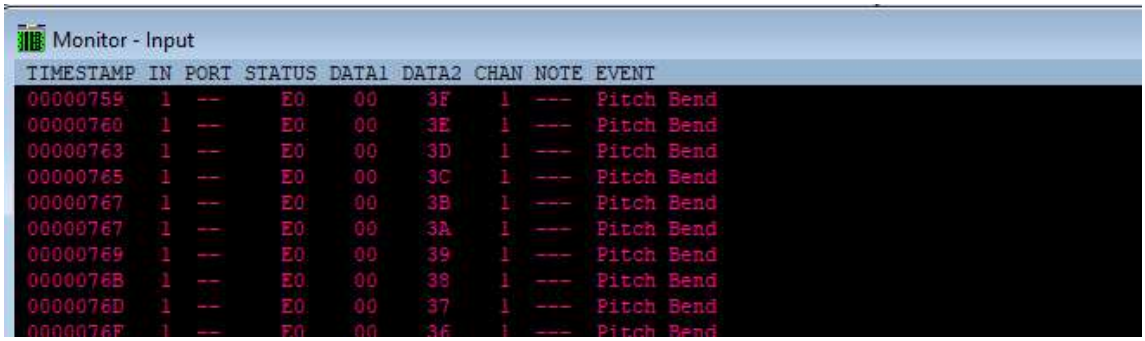


[...]

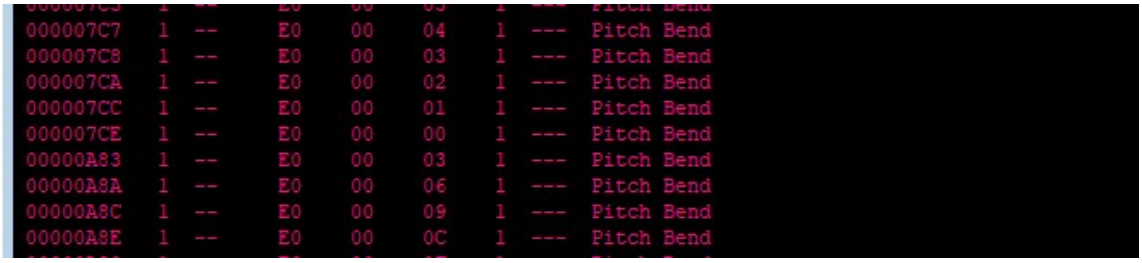


Com veiem, la rodeta del volum del Keystation Mini 32 comprèn un rang de 128 valors (0x00→0x7F), i tal i com hem vist a *Tipus de missatges MIDI*, gestiona el CC número 0x07, essent una comanda de Control Change (0xB).

Ara provarem el **Pitch Bend A** (botó <*PB* del Keystation Mini 32):



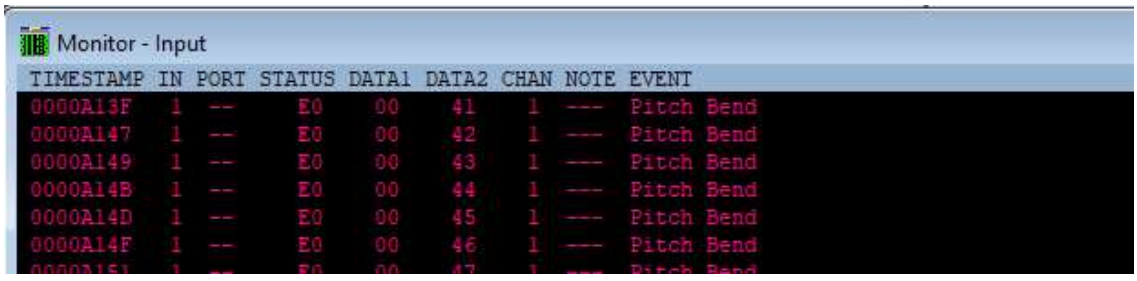
[...]



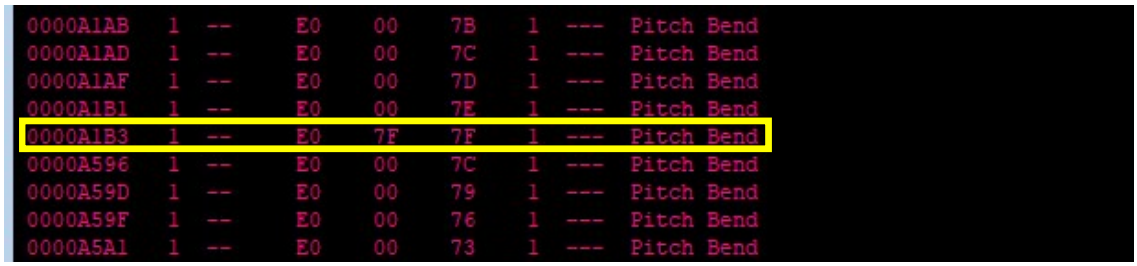
[...]

El valor per defecte del Pitch Bend, en aquest teclat, és 0x40 (64 en decimal). Al prémer el botó esmentat aquest valor comença a disminuir a una velocitat d'entre 1 i 3 mil·lisegons, i quan arriba a 0x00, torna a pujar al mateix ritme fins que de nou és a 0x40, on l'efecte s'esvaneix.

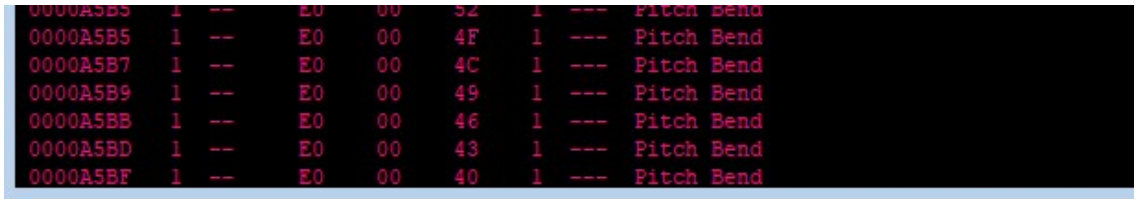
El **Pitch Bend B** (botó *PB*> del Keystation Mini 32) comença al valor de 0x40 i enlloc de disminuir, augmenta fins a 0x7F, punt en el que comença a disminuir fins que assoleix de nou el seu valor inicial.



[...]



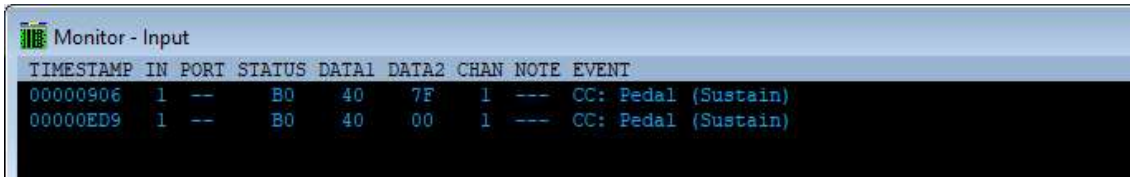
[...]



Observem que el nibble més significatiu (0xE) correspon satisfactòriament al Pitch Bend mencionat a la secció de *Tipus de missatges MIDI*. Tot i així, en els dos casos (Pitch Bend A i Pitch Bend B) que no es compleix al peu de la lletra el que havíem especificat en referència als format i valor dels bytes de dades. Allà, explicàvem que el Pitch Bend disposava de dos bytes de dades que actuaven conjuntament per formar un número de 14 bits que era el que representava el valor real de la comanda. Tanmateix, en el cas del Keystation Mini 32, sols disposem del segon byte de dades (DATA2 del MIDI-OX) per definir aquest valor, mentre que el primer (DATA1) sempre val 0x00 excepte quan el segon byte de dades es fica a 0x7F, moment en el que també passa a valdre 0x7F.

Per rebuscat que pugui semblar, a efectes pràctics no s'experimenta cap anomalia. Simplement, l'efecte del Pitch Bend en aquest teclat concret sols disposa de 2^7 valors per a definir-se, enlloc de 2^{14}, i el 0x7F del segon byte, que pot semblar arbitrari, sols ens indica que hem arribat al tope, que ja no hi ha valors més enllà del 0x7F.

A continuació, mirarem què passa si premem el **botó de *sustain*** (SUST):

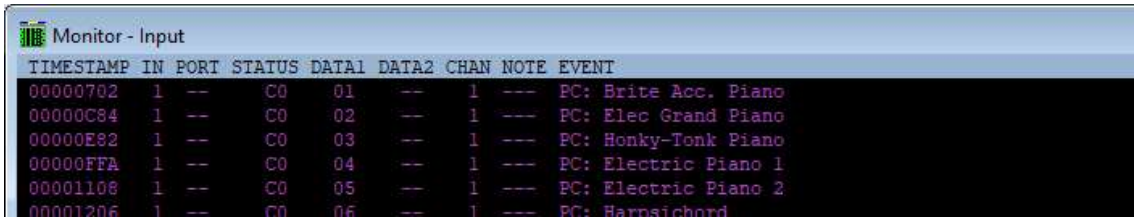


Com la rodeta de volum, aquest botó correspon a un CC amb el número 0x40, com podem deduir a través del nibble més significatiu del byte d'estat (0xB). El seu estat per defecte és inactiu (DATA2 = 0x00), així que quan premem el botó, s'activa (DATA2 = 0x7F). Quan tornem a prémer el seu valor retorna a 0x00. No hi ha valors entremig.

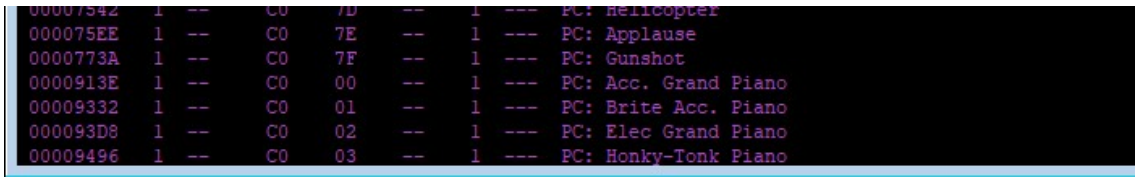
Finalment, analitzarem la resposta dels botons (+) i (-), definits com a OCT/DATA, que, segons el manual, permeten pujar i baixar l'octava del teclat en el programa seqüenciador i canviar d'instrument.

Ambdós botons responen com a Program Change (0xC), així que el que estan fent és canviar la pista o instrument actiu. Tanmateix, si connectem el teclat a l'ordinador i engeguem un programa seqüenciador, al tenir tots ells uns instruments propis, les comandes de canvi de pista resulten innecessàries, de manera que els botons (+) i (-) passen a regir el canvi d'octaves. Si premem (+), l'octava augmentarà i si premem (-), disminuirà.

Tot i així, per establir el canvi d'octaves, el Keystation Mini 32 fa servir comandes Program Change normals; són els programes seqüenciadors, els que ho interpreten diferent, de manera que MIDI-OX ens interpretarà les dades entrants d'ambdós botons (+) i (-) com a canvis d'instrument.

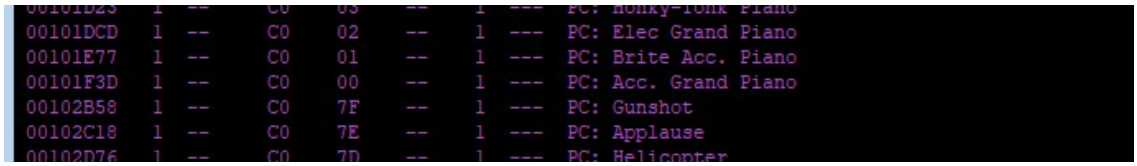


[...]



Com veiem, el botó (+) augmenta en u el número de l'instrument actiu. Com tots els dispositius MIDI entenent per defecte, aquest instrument és un Grand Piano. Hi ha 128 instruments, així que quan assolim l'instrument número 127 i tornem a prémer (+), l'instrument actiu retorna al número 0.

[...]



El botó (-) disminueix en u el número de l'instrument actiu. Quan tenim seleccionat l'instrument número 0 i tornem a prémer el botó, se'ns situa automàticament al 127.

Entès què és i com funciona l'estàndard del MIDI, passarem a explicar l'altre gran bloc funcional d'aquest projecte: l'estàndard S/PDIF.

4. QUÈ ÉS L'S/PDIF

S/PDIF és un estàndard de transmissió de senyals d'àudio digital a nivell de hardware impulsat per les companyies Sony i Philips (d'aquí ve el seu nom; Sony/Philips Digital Interface Format).

Va ser estandarditzat l'any 1985 com a *IEC 60958 type II*. Aquest protocol va cobrar gran èxit degut al cost reduït dels productes que l'empraven respecte als productes que feien servir altres protocols.

Malgrat que l'època daurada de l'S/PDIF va veure el seu final amb l'auge de l'HDMI, aquest sistema de transmissió d'àudio segueix sent molt comú i el podem trobar en qualsevol dispositiu audiovisual de casa nostra.

Com a característiques bàsiques, l'S/PDIF presenta dos canals d'àudio i permet reproduir so en estèreo. La seva informació està codificada en MIC (Modulació per Impulsos Codificats; veurem de què es tracta a continuació) i presenta un bon comportament davant de les interferències.

4.1 Tipus d'S/PDIF

Tenim dues maneres de fer funcionar l'S/PDIF; per mitjà de cables coaxials i per mitjà de la fibra òptica.

La coaxial és la versió més vella. La senyal es transmet a través d'un cable que conté un fil conductor a l'interior. La senyal es transmet elèctricament i, malgrat que resulti una opció més econòmica, queda més exposada a les pertorbacions electromagnètiques de l'entorn.

Per altra banda, la fibra consisteix en un cable òptic que transmet la informació mitjançant feixos de llum enlloc de senyals elèctriques. Això fa la transmissió immune a les pertorbacions electromagnètiques i el so resultant és més net. Tanmateix, aquestes millores respecte al cable coaxial fan de la fibra òptica una opció significativament més cara.

Tot i així, en aquest projecte ens decantarem per la fibra òptica i només emprarem un canal, doncs pel que es vol fer no ens és necessari reproduir l'àudio en estèreo.

4.2 Format de l'S/PDIF

L'S/PDIF no té una velocitat de transmissió definida per defecte, sinó que depèn de cada sistema en el que serveix. Això vol dir que l'S/PDIF treballarà a una freqüència o a una altra segons les especificacions de cada dispositiu. Per exemple, en els típics CDs, l'S/PDIF transmet les dades a una freqüència de 44.100 Hz, mentre que en els DAT (cintes d'àudio digital, vulgarment conegudes com a *cassette*) ho fa a 48.000 Hz.

Per norma general, la informació S/PDIF es transmet codificada en BMC (Biphase Mark Code). El BMC és un tipus de MIC, Modulació per Impulsos Codificats; un sistema de comunicació basat en cadenes d'informació binària. Els missatges BMC no són més que uns i zeros l'un després de l'altre, però en aquest cas concret la informació ve codificada per reduir el temps de transmissió i emetre dades més ràpidament.

Això vol dir que, per escoltar el so desitjat, primer caldrà descodificar tota la informació enviada.

DESCODIFICACIÓ DEL BMC (*Biphase Mark Code*)

La informació a transmetre via S/PDIF ve codificada en BMC. Per tal que soni l'àudio, s'ha de descodificar les dades de la manera següent:

Bit entrant (codificat) (1)	Bits sortint (descodificat) (2)
0	**11/00***
1	**01/10***

* Cada bit entrant es tradueix en una de les dues parelles de bits sortints (cèl·lula). Per saber quina de les dues és la correcta hem de veure el segon descodificat immediatament anterior. Si aquest bit era un 0, prendrem la combinació que comença per 1. Si aquest bit era un 1, prendrem la combinació que començava per 0. Aquest sistema impedeix que se succeeixin més de 2 bits iguals de manera consecutiva.

Per entendre-ho millor:

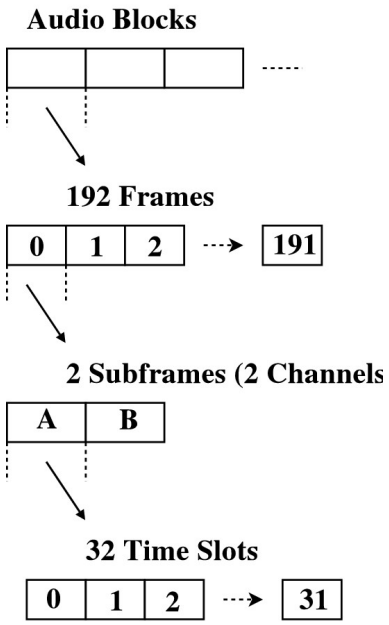
Input	Output
X 0	**X0 11**
X 0	**X1 00**
X 1	**X0 10**
X 1	**X1 01**

On X és un valor que no ens importa.

La informació S/PDIF consisteix en blocs d'àudio compostos per blocs de 192 trames cada una.Cada una d'aquestes trames consisteix en 64 bits.

Cada trama consta de dues subtrames de 32 bits cada una. La primera fa referència al canal A i la segona, al canal B. Això és útil si el que volem és reproduir àudio en estèreo. Si no volem, podem fer servir només el canal A.

Cada subtrama es divideix al seu temps en bits d'àudio i bits d'informació funcional S/PDIF.



Imatge 7. Esquema de l'estructura del protocol S/PDIF.

Per bits d'àudio entenem els bits que conformen el valor del so (en binari) a cada instant de temps, i per bits funcionals entenem els bits que serveixen per donar instruccions i característiques del so a reproduir.

El format d'aquestes subtrames és el següent:

Bits 0->3: *Preambles* o patrons de sincronització. Identifiquen la subtrama i sincronitzen la transmissió. No fan servir el BMC, sinó l'anomenat NRZ (non-return-to-zero. Tenim 3 *preambles* segons els 8 bits que obtindrem amb la descodificació:

Preamble "B": 11101000 (si el bit anterior era 0) o 00010111 (si el bit anterior era 1). Indica una paraula que conté informació pel canal A al principi del bloc de dades.

Preamble "M": 11100010 (si el bit anterior era 0) o 00011101 (si el bit anterior era 1). Indica una paraula que conté informació pel canal A, però que no està al principi del bloc de dades.

Preamble "W": 11100100 (si el bit anterior era 0) o 00011011 (si el bit anterior era 1). Indica una paraula que conté informació pel canal B.

Bits 4->7: bits d'informació auxiliars. Poden consistir en àudio de baixa qualitat, per comunicar-se entre dispositius o, simplement, per complementar l'àudio principal, consistent dels 20 bits següents, per augmentar la seva resolució en 4 bits.

Bits 8->27: informació d'àudio digital modulada en PCM i codificada en BMC. En cas de que l'àudio sigui de menys resolució que 20, els bits sobrants s'omplen automàticament amb zeros lògics.

Bit 28: "V", o bit de validació. Si val 0, el receptor reproduirà l'àudio. En cas contrari, l'ignorarà i silenciarà.

Bit 29: "U" o bit d'usuari. Informació addicional referent al so que ha de reproduir-se. Un bloc (192 trames) contindrà 192 bits d'usuari en total.

Bit 30: "C", o bit d'estat de canal. És el mateix que el bit 29, tot i que enlloc de fer referència a l'àudio, fa referència al canal.

Bit 31: "P", o bit de paritat.

En aquest projecte posarem un valor fix als bits auxiliars (4 a 7): “0000”. No necessitem donar instruccions extres a l’àudio ni atorgar-li més resolució. Fer-los servir de forma activa resultaria en un increment de la complexitat del programari que considerem innecessària.

El bit de validació (28) el mantindrem sempre a 1, ja que regularem quan ha de sonar l’àudio mitjançant el Note On i Note Off del MIDI.

El bit d’usuari (29) el mantindrem a 0, ja que no requerim informació addicional per gestionar els blocs d’informació d’àudio.

El bit d’estat (30) el mantindrem a 0. El projecte està pensat per reproduir una única trama d’àudio. Dos canals ens serien útils per reproduir so en estèreo, per exemple, si tinguéssim dues fonts emissores. En aquest cas sols tenim un teclat que no fa distinció del canal a emprar. Mantenint el bit d’estat a 0 ens activarà permanentment el Channel A de l’S/PDIF, que és per on sonarà la música que toquem.

El bit de paritat (31) serà ‘1’ o ‘0’ en funció de la paritat de cada trama.

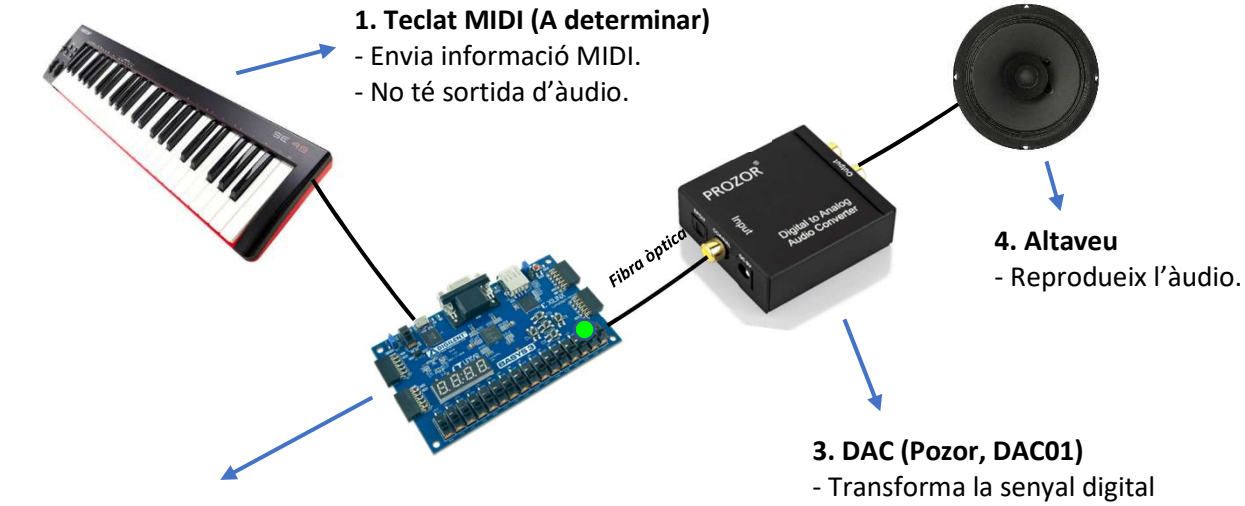
5. HARDWARE

Establir el hardware del piano va resultar un dels punts més complicats del projecte, doncs es va haver de batallar per comunicar un dispositiu amb un altre, impediments desconeguts que van requerir la inclusió de nous dispositius i algun invent.

Com veurem a continuació, a l'inici del projecte es va proposar un esquema general que, degut als mencionats problemes, va patir més d'una modificació al llarg dels mesos. En les següents pàgines seguirem, cronològicament, el per què de cada una d'aquestes modificacions.

5.1 Idees inicials

La idea original d'aquest projecte consistia en el desenvolupament d'un piano MIDI amb sortida S/PDIF mitjançant una FPGA programada amb VHDL. Com que és molta informació de cop, l'esquema següent intenta resumir-ho:



Imatge 8. Esquema dels components originals del projecte.

Tanmateix, per vàries raons, aquesta idea original va mutar a poc a poc, com a mínim, pel que fa a la part de la comunicació entre teclat i FPGA. Malgrat que al final el resultat

fos l'esperat, durant el trajecte va haver-hi molts canvis que van fer replantejar les maneres d'arribar-hi.

Aquestes raons són les que s'expliquen a continuació.

Nota: en les següents pàgines es fa referència només a la primera part del circuit (del teclat a l'FPGA), ja que de l'FPGA a l'altaveu no va haver-hi modificacions de cap mena.

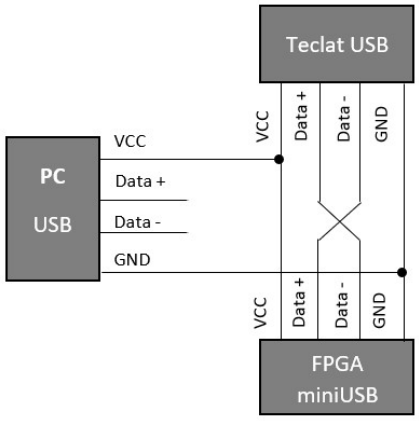
a) Alimentació dels dispositius.

El teclat adquirit, el Keystation Mini 32 de M-AUDIO, no disposava de bateria o d'un segon cable d'alimentació. En el seu lloc, aquest teclat havia d'alimentar-se forçosament pel mateix canal pel que emetia les dades en sèrie (MIDI).

Imatge 9. Keystation Mini 32.

Com a contramesura es va estudiar a veure si era possible que la via de comunicació teclat-FPGA també pogués servir d'alimentació pel teclat. Però el port sèrie amb el que havia de comunicar-se amb el teclat permetia el pas d'informació, però no alimentar el teclat.

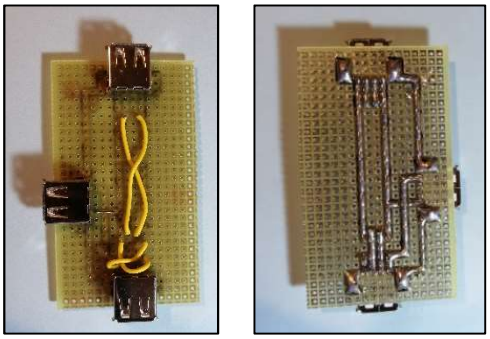
Aquest fet va obligar a alimentar tant el teclat com l'FPGA de forma independent. Com que l'únic port del teclat era USB i el port sèrie de l'FPGA també, es va dissenyar el següent circuit:



Imatge 10. Esquema del circuit d'alimentació del teclat i l'FPGA.

Amb aquest circuit, un port USB de l'ordinador serviria per alimentar tant al teclat com l'FPGA. Ambdós components quedarien alimentats mentre establissin la comunicació sèrie via UART.

Per muntar el circuit van requerir-se 3 connectors USB femella, una placa Repro, un soldador i una mica d'estany i uns quinze centímetres de cables pelables. El resultat és el següent:



Imatge 11. Circuit d'alimentació del teclat i l'FPGA.

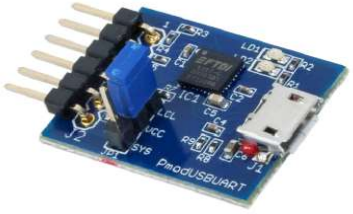
Malgrat que tan el teclat com l'FPGA quedessin correctament alimentades a través de l'ordinador, es va fer un programa de prova per comprovar si la informació sortint del teclat la rebia bé l'FPGA a través d'una UART. No funcionava.

Descobriments posteriors demostrarien la necessitat de més canvis en l'estructura del projecte.

b) Comunicació USB - UART

La informació USB sortint del teclat no estava preparada per ser reconeguda per la UART que implementàvem a l'FPGA. Es necessitava un dispositiu capaç de convertir la informació USB en dades sèrie.

La solució va passar pel PmodUSBUART de Digilent; un dispositiu capaç de transformar dades confuses de la interfície USB a informació comprensible per una UART i viceversa fent servir un xip FTDI.



Imatge 12. PmodUSBUART de Digilent.

El PmodUSBUART es va muntar entre el teclat i l'FPGA, fent ús d'un dels connectors femella USB de la placa Repro. Però completat el circuit i executats de nou els programes de test la comunicació entre el teclat i l'FPGA semblava no existir.

Això es deuria a un tercer i últim punt essencial que implicaria un darrer canvi en l'estructura del projecte: el "feedback" del MIDI.

c) Feedback del MIDI

Veient que el programa seguia sense funcionar, es va prendre un oscil·loscopi per comprovar, senyal per senyal i dispositiu per dispositiu, què era el que fallava.

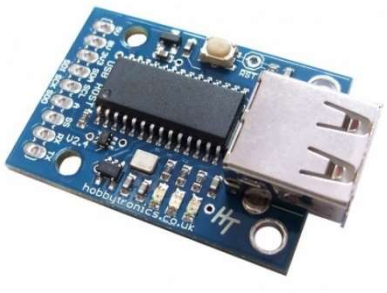
Es va concloure que el teclat no enviava informació en sèrie endavant. Sense aquesta informació, el programa no podia sintetitzar res, i la música no sonava perquè no es generava la senyal des de bon principi.

Per comprovar que el teclat no estava espatllat, es va connectar a un dels ports de la placa Repro, i a l'extrem oposat es va connectar un cable directe cap a un dels ports USB de l'ordinador.

D'aquesta manera, quan es premés una tecla del teclat, es podria veure si s'enviava informació directament a l'ordinador punxant la via d'estany de la placa. A la pantalla de l'oscil·loscopi ja no apareixia una línia tènue de soroll, sinó ones difuses que evidenciaven que el teclat s'estava comunicant amb l'ordinador.

Resulta que el MIDI no emet senzillament la informació, sinó que requereix d'un feedback. Aquest feedback permet sincronitzar el controlador MIDI (en aquest cas, el teclat) i el dispositiu de síntesi d'àudio (en aquest cas, l'ordinador).

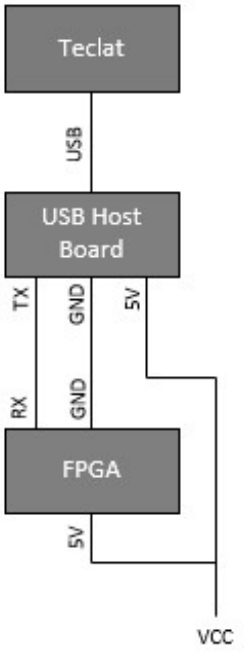
La forma més eficaç que va trobar-se per aconseguir aquesta sincronització entre el teclat i l'FPGA va trobar-se en un nou dispositiu: l'USB Host Board de Hobbytronics.



Imatge 13. USB Host Board de

Aquest dispositiu basa el seu funcionament en el microcontrolador 24FJ64GB002 (l'IC allargat que podem veure a la foto). Malgrat que aquest microcontrolador vingués verge al principi, la pròpia pàgina de Hobbytronics disposava d'un manual i els enllaços corresponents per instal·lar-li un dels molts softwares que la pròpia empresa oferia. Aquests softwares atorguen al microcontrolador una funcionalitat específica, una d'elles, permetre la conversió segura d'informació entre les interfícies MIDI i USB.

Bàsicament, aquest dispositiu amb el software mencionat, faria que el teclat i la FPGA s'entenguessin tal i com s'entenien el teclat i l'ordinador, doncs l'ordinador ja disposa dels mecanismes necessaris per comprendre el protocol MIDI.



Imatge 14. Esquema final del hardware del projecte .

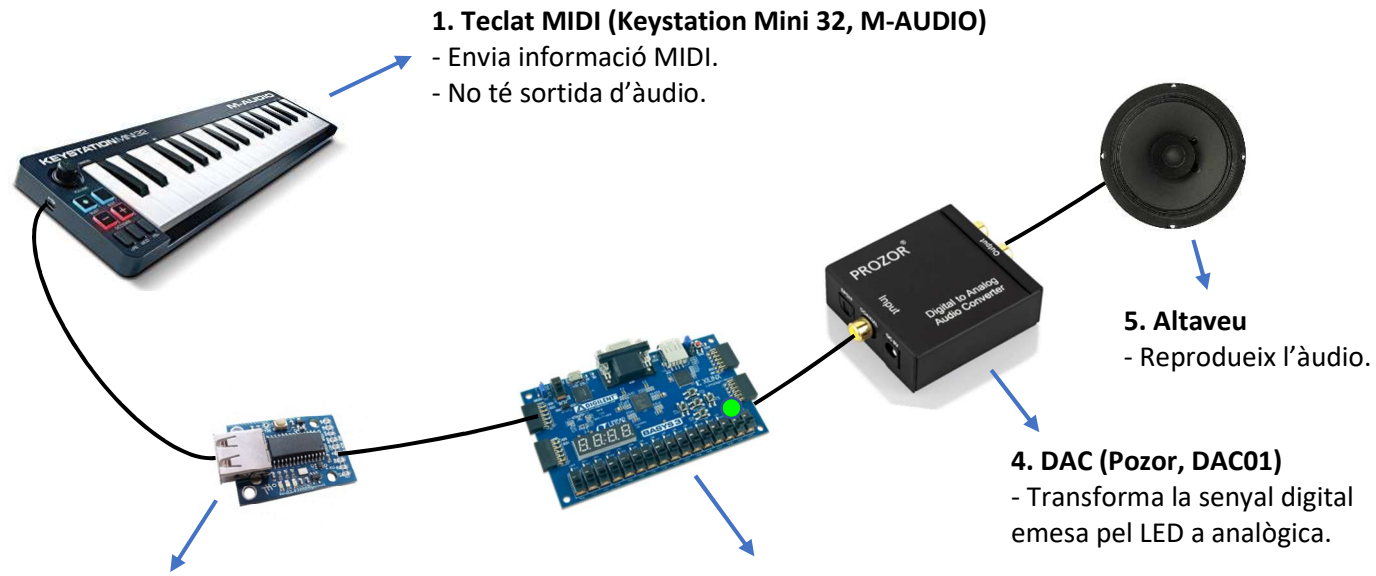
Per instal·lar l'USB Host Board al projecte es va haver de modificar l'esquema de la manera que es mostra a la dreta.

Veiem que la placa Repro ja no és necessària, ja que per alimentar el teclat ara tenim l'USB Host Board. Tampoc necessitem el PmodUSBUART, ja que l'USB Host Board ja inclou la funció de tractar les dades USB. Sens dubte, aquest nou dispositiu va simplificar molt el projecte.

Els programes de test es van tornar a carregar a l'FPGA. Finalment, els resultats van ser els esperats. Cada dispositiu feia la seva feina, i al prémer qualsevol tecla del teclat, el so es formava i reproduïa correctament.

5.2 Estructura final del projecte

El hardware del projecte, doncs, amb tota la informació donada a *5.1 Idees inicials*, va quedar de la manera següent:



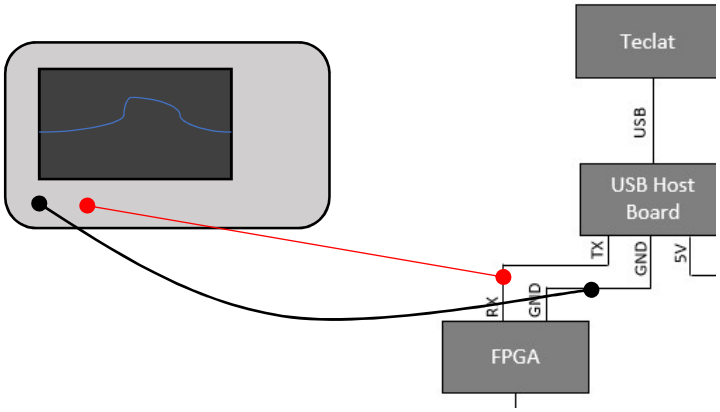
Imatge 15. Esquema complet del hardware final del projecte.

5.3 Visualització dels missatges MIDI

Un dels passos fonamentals per conèixer com es transmetien les dades MIDI i comprovar el bon funcionament del teclat va ser observar la comunicació en sèrie del Keystation Mini 32.

Veure com es transmetia cada missatge MIDI seria essencial per desenvolupar un programari capaç d'absorbir aquesta informació, sintetitzar-la i convertir-la a l'estàndard S/PDIF.

Per observar aquests missatges es va emprar un oscil·loscopi, connectant-se la sonda a la senyal TX de l'USB Host Board i referenciant-se a la massa del mateix dispositiu.

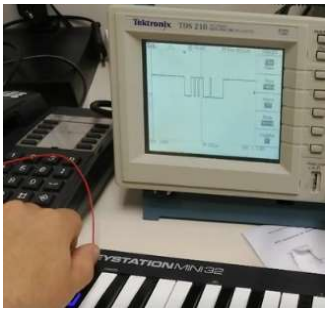


Imatge 16. Presa de senyals amb l'oscil·loscopi.

Un cop tot connectat i configurat el *trigger* de oscil·loscopi, al prémer les tecles del Keystation Mini 32 es podien observar els corresponents bytes d'*status* i de dades, i es va poder comprovar que el teclat funcionava perfectament i que els missatges MIDI s'ajustaven a l'explicació donada al capítol 3.

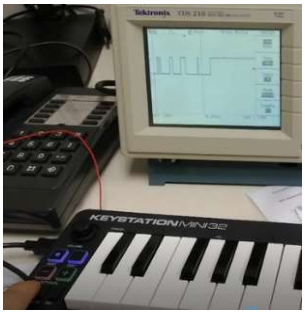
A continuació es mostren alguns exemples de les tecles premudes del teclat i les senyals obtingudes:

1. Rodeta del volum a mínim



2. Rodeta de volum a màxim

3. Pitch esquerre

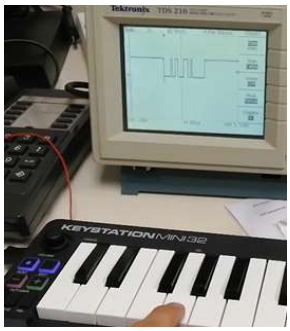


4. Pitch dret

5. Tecla D#2 / Transpose

6. Tecla G2 (Velocitat alta)

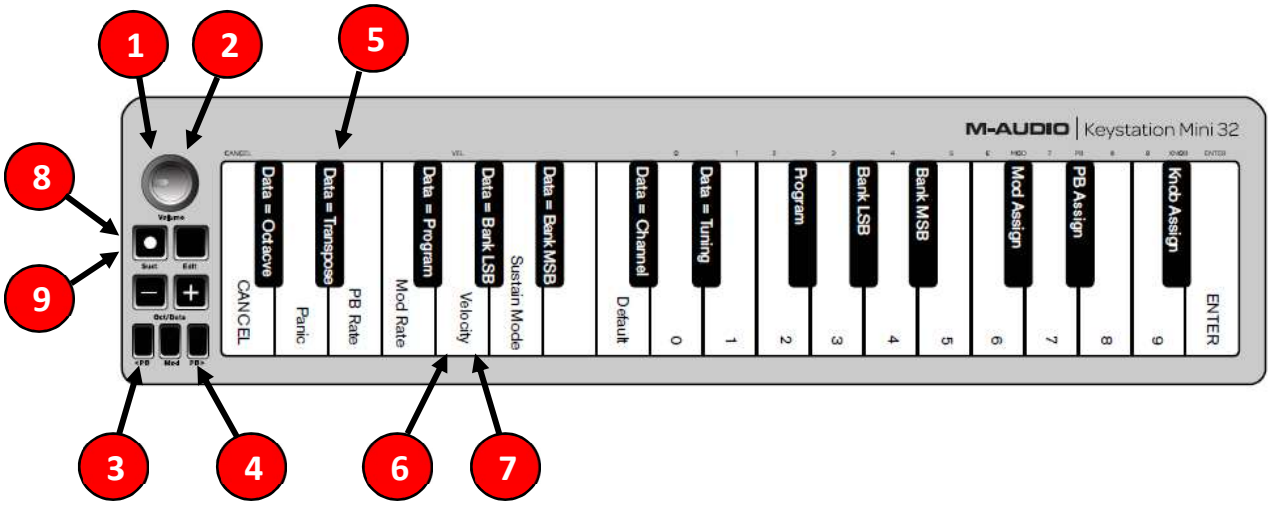
7. Tecla G2 (Velocitat baixa)



8. Sustain ON

9. Sustain OFF

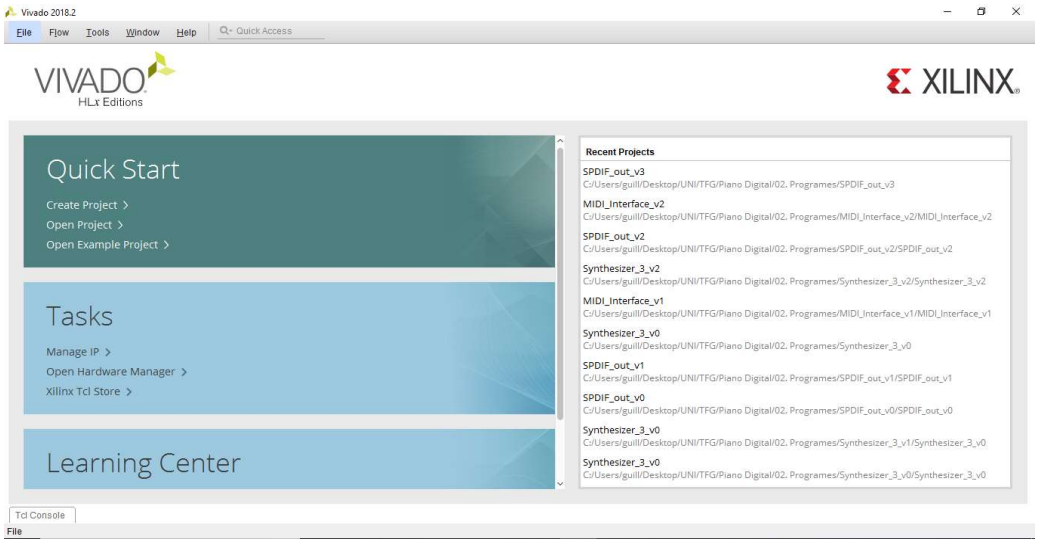
Imatge 17. Nou missatges MIDI i els seus respectius bytes.



Imatge 18. Les tecles del Keystation Mini 32. Els números indiquen la foto a la que correspon la senyal.

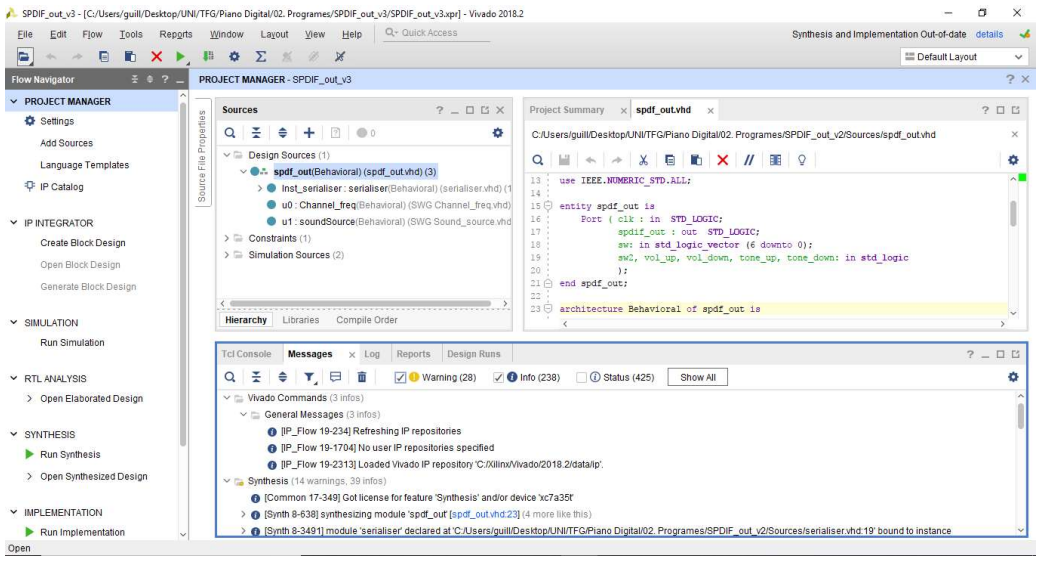
6. PROGRAMARI

El programari d'aquest projecte es basa en el llenguatge de baix nivell VHDL, i s'ha desenvolupat a través de l'eina Vivado Design Suite 2018.2 de Xilinx.



Imatge 19. Portal del Vivado 2018.2 de Xilinx.

El Vivado no només permetia programar amb certa comoditat sinó que oferia la possibilitat de sintetitzar i analitzar els programes desenvolupats i establir comunicació amb l'FPGA. En poques paraules, amb el Vivado es podria crear el programari que transformaria els missatges MIDI en trames S/PDIF i a més a més ens deixaria instal·lar-lo a l'FPGA.



Imatge 20. Interfície de programació del Vivado 2018.2 de Xilinx.

El desenvolupament del programari del projecte va constar de tres fases fonamentals, per bé que en les següents pàgines ens centrarem en les dues finals:

a) **Fase d'aprenentatge**

   Aquesta fase engloba totes les hores necessàries per aprendre el llenguatge VHDL i dominar l'eina del Vivado. Va comportar una important etapa de documentació i una altra de desenvolupament de programes experimentals en la que va resultar essencial disposar de l'FPGA per comprovar-ne el funcionament.

b) **Fase de testos**

   Una vegada dominats el llenguatge VHDL i el Vivado, la creació de programes experimentals no es va aturar. En el seu lloc els nous programes van enfocar-se a provar dels dispositius per entendre com es comunicaven i treure'n el màxim partit.

   En el capítol següent, *6.1 Fase de testos*, entrarem en detall en cadascun d'aquests programes per avaluar el funcionament de cada dispositiu.

c) **Desenvolupament del programa**

   El desenvolupament del programa final va servir-se de les fases anteriors. Es van aprofitar els programes de test, perfeccionats al llarg de les setmanes, per muntar un programa capaç de sintetitzar la informació MIDI del teclat en trames S/PDIF.

   Entrarem a aquesta fase amb més detall en el capítol *6.2 Programa final*.

6.1 Fase de testos

La fase de testos comprèn el desenvolupament dels programes destinats a comprovar com rebre els missatges MIDI del teclat, sintetitzar aquests missatges per obtenir característiques de so, crear ones sonores en base a aquestes característiques i transformar-les en trames S/PDIF. En les següents pàgines s'avaluaran els tres programes principals emprats per aquests propòsits.

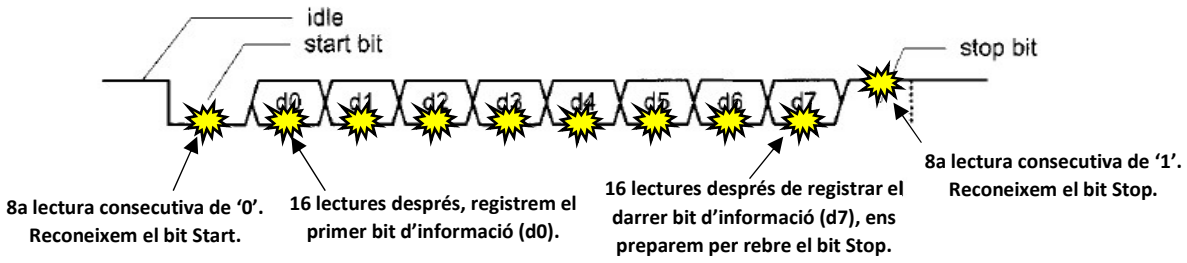
6.1.1 MIDI Interface

MIDI_Interface és un programa que té per objectiu captar els missatges MIDI enviats pel teclat mitjançant una UART i sintetitzar-la per obtenir la informació del so a reproduir.

Consta de tres blocs principals:

- **UART**

  La UART (Universal Asynchronous Receiver Transmitter) és un circuit de comunicació de dades en sèrie, com diu el seu nom, de manera asíncrona.
  Disposa de dos canals, un de transmissió (TX) i un de recepció (RX). Per funcionar correctament, l'emissor i el receptor han de definir una velocitat de treball per poder-se entendre. En aquest programa només es contempla la funció de recepció de la UART, doncs rebem els missatges MIDI del teclat però no necessitem retornar-li cap mena d'informació.
  La transmissió (TX) es fa mitjançant un registre de desplaçament. Es carreguen vuit bits d'informació encapçalats per un bit Start ('0') i acabats per un bit Stop ('1') i s'envien bit per bit al receptor. El receptor, que té una velocitat de mostreig 16 vegades superior a la velocitat a la que es transmeten les dades, quan detecta el bit Start inicia un procés per a registrar cada un dels 8 bits d'informació següents, fins que assoleix el bit Stop.
  Per registrar cada un dels bits, el present programa ho fa de la següent manera: L'estat per defecte de cada línia de comunicació és '1' per defecte. El receptor llegeix el valor del canal RX a una freqüència de 500kHz. Així doncs, si no es transmet cap informació, la UART estarà llegint 500.000 bits '1' cada segon. Tanmateix, si decidim enviar un missatge MIDI i s'inicia la transmissió de dades, aquest valor per defecte canviarà. Si durant 8 lectures consecutives el valor llegit és '0', voldrà dir que ens trobem a la meitat del bit Start, i això significa que s'ha iniciat la transmissió de dades en sèrie. Per tal de rebre aquesta informació correctament, la UART haurà de seguir llegint el canal RX a 500kHz, però el valor del primer bit d'informació el prendrà a la 16ena lectura. Així ens assegurem que ens trobem a la meitat del primer bit. Aquest procés es repetirà fins que es detecti el bit Stop i retornem al valor per defecte ('1') de la línia de comunicació. Els valors presos, la UART els emmagatzema en un registre que al final de la transmissió enviarà al dispositiu al que serveix.



Imatge 21. Recepció de bytes d'informació amb una UART.

Estudiat el funcionament d'aquesta UART, es va assegurar que la informació procedent de l'USB Host es rebia sense problemes. Amb aquest pas assolit es podrien reconstruir els missatges MIDI donats pel teclat. La gestió dels mateixos correspondria a la següent etapa del programa.

- **MIDI Core**

  S'ha anomenat MIDI Core a l'etapa de gestió dels missatges MIDI. Aquest sub-programa s'encarrega de rebre els bytes de dades recopilats per la UART, identificar de quin tipus de missatge MIDI es tracten, desglossar-los en informació útil i transferir les dades resultants a la següent etapa.

  Sabent que cada missatge MIDI es compon primer d'un byte d'estat i després d'un byte de dades, és senzill desxifrar el significat de cada missatge.

  El primer byte, el d'estat, diu la instrucció, i el segon, el de dades, especifica les condicions de la instrucció.

  Mitjançant una màquina d'estats podem desglossar i entendre cada missatge, i enviar les dades correctes a l'etapa següent.

  Aquestes dades resultants són:

  a) *Midi_ch:*

  Canal al que afecta el missatge MIDI actual (del 0 al 15). En aquest projecte sols emprarem el canal 1, doncs no en necessitem més. La resta s'ha deshabilitat per reduir la complexitat del programa.

  b) *Midi_note:*

  En el cas que el missatge MIDI es tracti d'una nota, indica la nota a la que equival (de la 0 a la 127).

  c) *Midi_velo:*

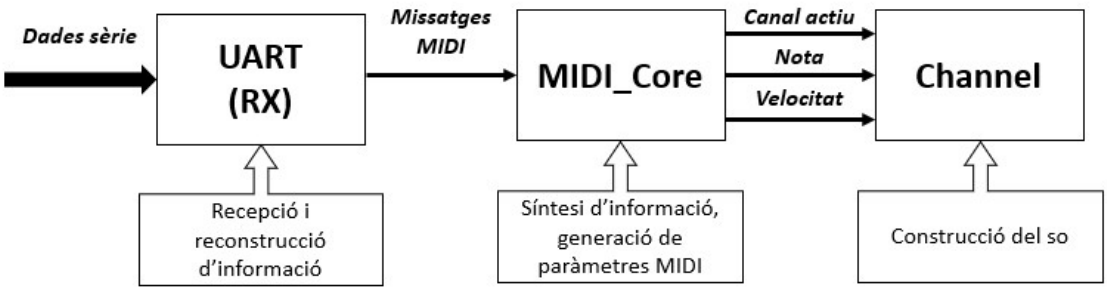
En el cas que el missatge MIDI faci referència a una nota, indica la velocitat de premuda de la tecla (de 0 a 127). En aquest projecte, com que la velocitat la controlarem a través de l'FPGA (tant el nivell del volum com l'activació i desactivació del so) s'ha establert que si la velocitat és 0 la nota no ha de sonar, i si la velocitat és major de 0, el volum de la nota serà el marcat mitjançant l'FPGA.

- **Channel**

  Al sub-programa de Channel és l'encarregat de rebre les instruccions del MIDI_Core i obeir-les, donant forma al so a reproduir.

  Tanmateix, com que el programa MIDI_Interface té per objectiu provar que els missatges MIDI són rebuts i compresos correctament, Channel pren una funcionalitat de debug. 7 LEDs de l'FPGA es van destinar a simular els 7 bits de dades que descriuen la nota actual. Un botó de l'FPGA servia per mostrar el valor binari de la última nota premuda mitjançant els 7 LEDs.

Si els LEDs encesos es corresponien amb el valor binari de la nota premuda al teclat, és que cadascuna de les diferents etapes de MIDI_Interface actuava correctament. Després de diverses proves i correccions, el programa funcionava a la perfecció.



Imatge 22. Esquema de MIDI_Interface.

6.1.2 SWG

El SWG (Soundwave Generator) genera sonores a partir d'un registre de valors ordenats i una freqüència i amplitud donades.

Per registre de valors ordenats entenem una seqüència d'enters que, mostrats en una gràfica, conformen la forma d'ona del so d'un instrument.

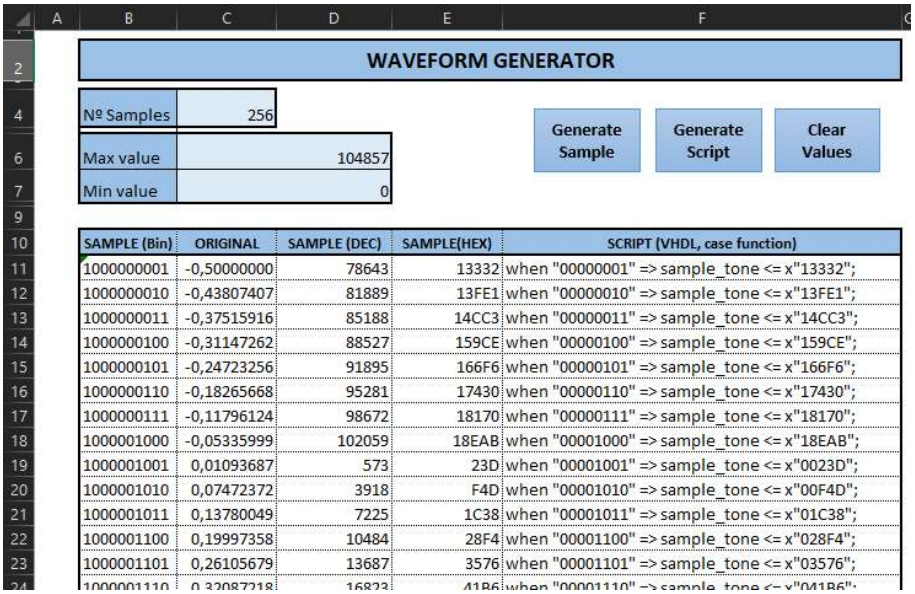
La freqüència és la rapidesa amb la que es reprodueix la seqüència. És un paràmetre variable que definirà la nota. Com més alta sigui la freqüència, més aguda serà la nota, i com més lenta sigui la freqüència, més greu serà la nota.

L'amplitud, que també és un paràmetre variable, permet augmentar o disminuir el volum del so. Com més amplitud, més volum, i viceversa.

Per generar el so (instrument, nota i volum), el desenvolupament del SWG va consistir en dues parts: creació de les formes d'ona i desenvolupament del programa en VHDL.

Per crear les formes d'ona (instruments) les macros de l'Excel van resultar una eina molt eficaç. Donada una equació que definís la forma d'ona de l'instrument desitjat, un número de punts amb la que dividir-la i un valor màxim i mínim que comprenguessin els punts de més i menys amplitud, respectivament, es construïa de forma automàtica un registre de valors ordenats.

Aquesta macro, anomenada SWG, es va desenvolupar i millorar diverses vegades per aconseguir el resultat més net i precís possible. A més a més, la darrera actualització permetria generar l'script en VHDL amb el que s'implementaria la forma d'ona al programa.



WAVEFORM GENERATOR

Nº Samples	256
Max value	104857
Min value	0

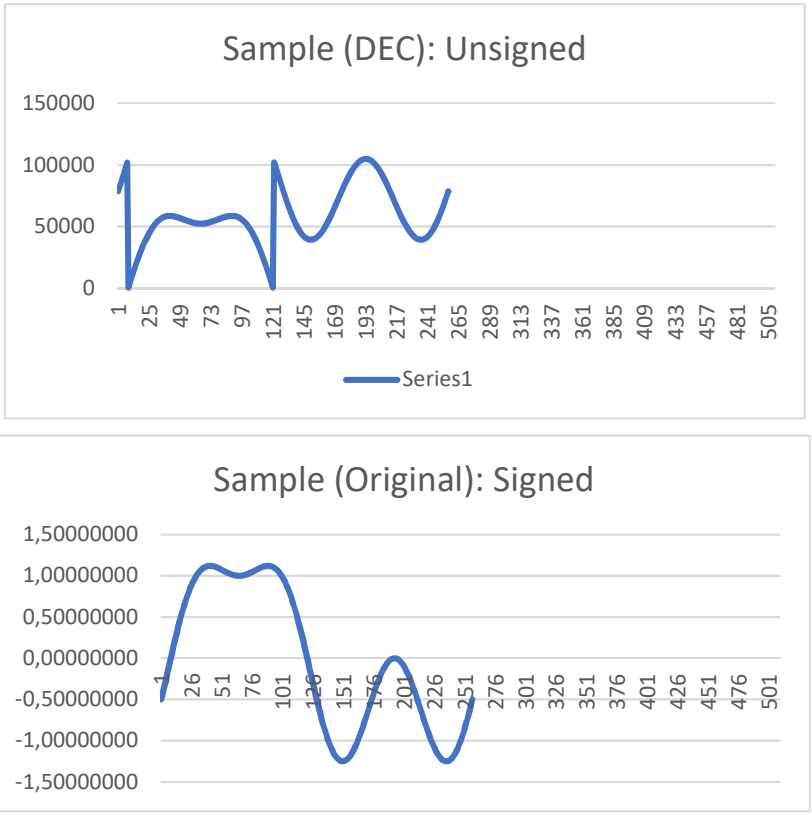
Generate Sample | Generate Script | Clear Values

SAMPLE (Bin)	ORIGINAL	SAMPLE (DEC)	SAMPLE(HEX)	SCRIPT (VHDL, case function)
1000000001	-0,50000000	78643	13332	when "00000001" => sample_tone <= x"13332";
1000000010	-0,43807407	81889	13FE1	when "00000010" => sample_tone <= x"13FE1";
1000000011	-0,37515916	85188	14CC3	when "00000011" => sample_tone <= x"14CC3";
1000000100	-0,31147262	88527	159CE	when "00000100" => sample_tone <= x"159CE";
1000000101	-0,24723256	91895	166F6	when "00000101" => sample_tone <= x"166F6";
1000000110	-0,18265668	95281	17430	when "00000110" => sample_tone <= x"17430";
1000000111	-0,11796124	98672	18170	when "00000111" => sample_tone <= x"18170";
1000001000	-0,05335999	102059	18EAB	when "00001000" => sample_tone <= x"18EAB";
1000001001	0,01093687	573	23D	when "00001001" => sample_tone <= x"0023D";
1000001010	0,07472372	3918	F4D	when "00001010" => sample_tone <= x"00F4D";
1000001011	0,13780049	7225	1C38	when "00001011" => sample_tone <= x"01C38";
1000001100	0,19997358	10484	28F4	when "00001100" => sample_tone <= x"028F4";
1000001101	0,26105679	13687	3576	when "00001101" => sample_tone <= x"03576";

Imatge 23. Interfície de l'Excel SWG. L'equació s'entra directament a la macro. Amb els botons blaus es genera la forma d'ona i l'script d'implementació VHDL.

Es van generar tres formes d'ona de mostra en la realització d'aquest projecte: sinusoidal, piano i gaita.

La macro SWG mostra també la gràfica de punts de la ona. Les gràfiques de baix corresponen a la forma d'ona de la gaita.



Imatge 24. Forma d'ona de la gaita. Signed i Unsigned són els mateixos valors visualitzats diferents. Unsigned són els valors que el programa VHDL entén i processa. Signed són els valors que conformen la forma d'ona que nosaltres entenem.

Per desenvolupar el SWG en VHDL, es van programar dos blocs funcionals:

a) Sound_source

L'objectiu del Sound_source és determinar l'instrument a tocar i el seu volum. Aquest bloc conté els tres instruments (registres de punts ordenats) generats per la macro. Té assignats a l'FPGA dos botons per pujar o baixar d'instrument.

Segons l'instrument seleccionat, la variable *sample_tone* (senyal principal del bloc Sound_source) recorre els punts d'una forma d'ona o una altra.

Cada instrument disposa d'un total de 256 punts. *Sample_tone* pren per valor cada un d'aquests punts de forma ordenada però a una freqüència variable (determinada pel següent bloc del programa).

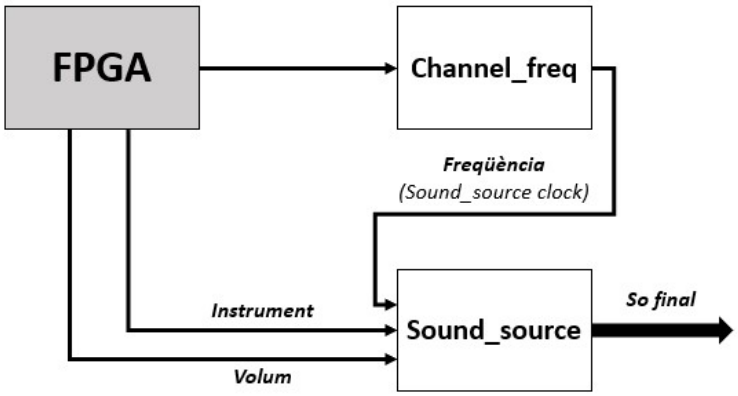
Abans de transmetre's, el valor de *Sample_tone* es multiplica pel volum. El resultat d'aquesta multiplicació no farà variar l'aparença de la forma d'ona (i per tant, el timbre de l'instrument) sinó el volum. Tenim 6 nivells de volum (de 0 a 6)

que controlarem mitjançant dos polsadors de l'FPGA. A 0, el volum es considera nul i per tant no sentirem res a l'altaveu.

b) Channel_freq

Aquest bloc s'encarrega de definir la velocitat a la que *Sampe_tone* canvia de valor. Disposa d'un *clock* intern que varia en funció de la nota premuda. Dins del bloc *Channel_freq* existeix un registre on cada nota del teclat té assignada una freqüència.

Com més aguda sigui la nota premuda, més alta serà la freqüència activa i més ràpid avançarà el *clock*. Això farà que *Sample_tone* avanci més veloçment a través del registre de punts; segons la rapidesa amb la que es reprodueixi la forma d'ona a *Sound_source*, més agut o greu serà el so generat.



Imatge 25. Esquema de SWG.

Per si sol, el SWG no podria provar-se fins que es desenvolupés l'SPDIF_Out, que és el tercer i últim programa de prova.

6.1.3 SPDIF Out

SPDIF_Out és el darrer programa de la fase de proves i està destinat a convertir el so generat pel SWG a l'estàndard S/PDIF.

Per desenvolupar l'SPDIF_Out es va incloure el SWG com un dels blocs fonamentals juntament amb el sub-programa *Serialiser*. Com que ja s'ha explicat el funcionament del SWG, ens centrarem en el *Serialiser*.

El *Serialiser* és l'encarregat de crear i enviar les trames S/PDIF en base al valor que en cada instant de temps li proporciona el SWG. Consta de quatre passos elementals: lectura, establiment de paràmetres S/PDIF, generació de la trama i enviament. Per no sobreescriure's valors enmig d'una operació, existeix un *clock* intern (*Timebase*), que notifica al sistema quan s'està processant una trama. Fins que aquest *clock* no envia un *tick* de validació, no es pren un nou valor del SWG.

a) Lectura

La lectura s'inicia sempre que el *Timebase* envia un *tick* de validació notificant que la trama anterior s'ha acabat de processar.

El *Serialiser* obté el valor del so cedit pel SWG, guardant-lo en una nova variable de vint bits abans d'enviar-la al següent pas.

b) Establiment de paràmetres

En aquest pas es defineixen les característiques de la trama S/PDIF; s'escull el *preamble* i s'estableix el valor dels bits auxiliars, del bit de validació, el d'estat de canal i d'usuari.

Per bé que podríem haver emprat els 4 bits auxiliars per a augmentar la resolució de l'àudio, de 20 a 24 bits, al final els hem deixat a 0 per dos motius:

1. Complexitat de la transmissió, ja que hauríem de donar missatges específics via *preambles* per tal d'habilitar els bits auxiliars per a incrementar la resolució.
2. El so que volem emetre és molt senzill. Consisteix bàsicament en formes d'ona simples que amb 8 bits ja podrien sonar bé. Considerem innecessària augmentar la resolució a 24 bits.

El bit de validació es mantindrà sempre a '0', ja que a '1' ens silenciaria la sortida.

El de canal, també el mantindrem a '0', doncs com que només es fa servir el Channel A de l'S/PDIF, el mantindrem actiu permanentment. El bit d'usuari també es deixà a '0', ja que no hi ha missatges de supertrama que puguin resultar útils en aquest projecte.

El bit de paritat es calcularà una vegada construïda la trama, doncs depèn del valor de la resta de bits de la trama (a excepció del *preamble*, que no es compta en aqueta operació).

c) Generació de la trama

Al generar-se la trama, es prenen els paràmetres definits al pas anterior a més del valor del so. Es distribueixen tots els bits d'informació en una cadena de 32 bits en total, ordenada segons l'estàndard S/PDIF (veure capítol *4.2 Format de l'S/PDIF* per més detalls).

d) Enviament

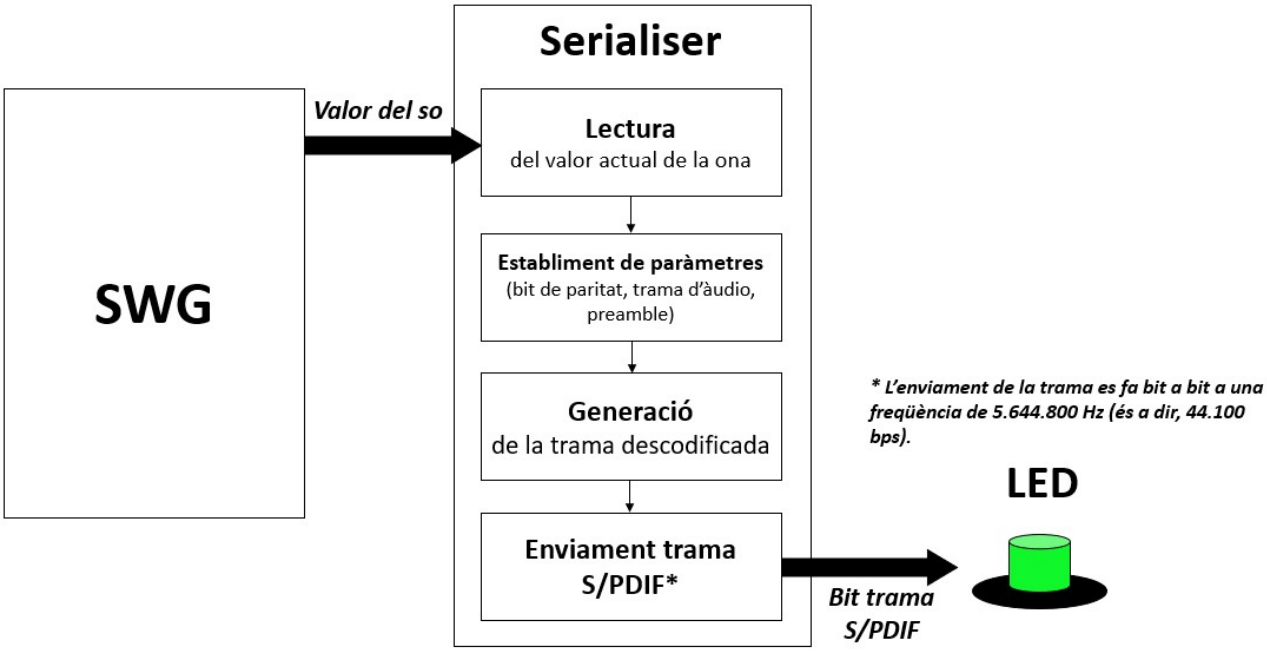
Construïda la cadena, s'envia bit per bit a 44.100 *bps* a un dels LEDs de l'FPGA. Aquest LED emet la informació de la manera següent:

Si el bit actual de la trama és 1, el LED s'encén.

Si el bit actual de la trama és 0, el LED s'apaga.

El LED s'encén i s'apaga tan ràpidament que a simple vista no ho podem distingir; sempre el veurem encès. Així doncs, per comprovar el funcionament de l'SPDIF_Out cal situar l'extrem d'un cable de fibra òptica apuntant al LED, i l'altre extrem connectar-lo a un DAC.

El DAC converteix la senyal digital S/PDIF a àudio analògic, i connectat al seu temps a un altaveu, el so es reprodueix correctament.

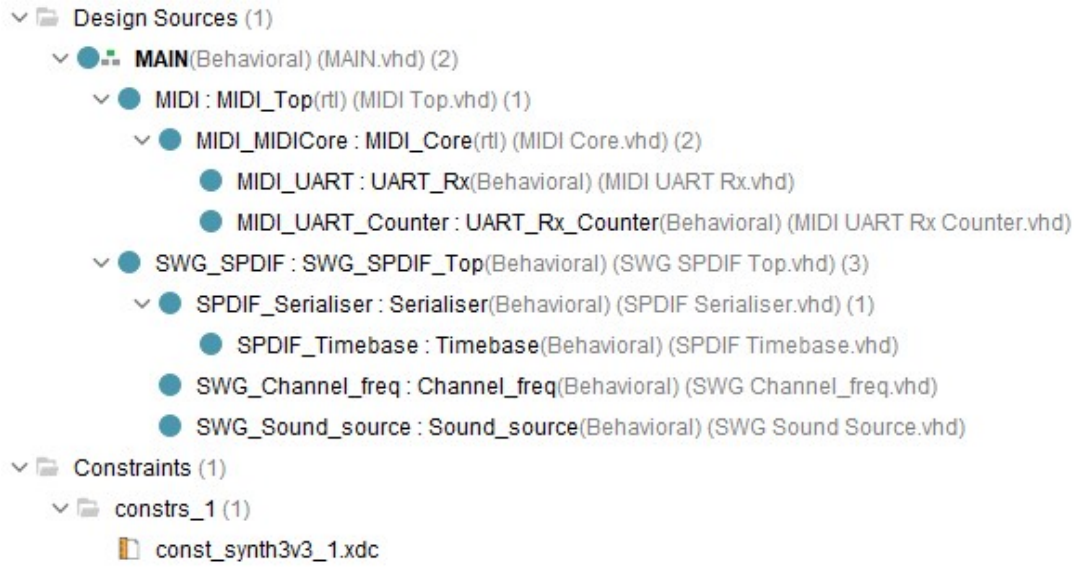


Imatge 26. Esquema de SPDIF_Out.

6.2 Programa final: Synthesizer

El programa definitiu, anomenat Synthesizer, és, resumidament, la unió de les entitats MIDI_Interface, SWG i SPDIF_Out. Òbviament, per aconseguir aquesta unió va caldre modificar els tres blocs per adaptar-ne les connexions i revisar ports i senyals.

Analitzarem el programa Synthesizer, primerament, contemplant la vista general dels diferents sub-programes implicats estructurats jeràrquicament:



Imatge 26. Esquema del programa Synthesizer.

MAIN

Com veiem a dalt de tot, tenim una entitat controladora anomenada MAIN.

MAIN enllaça les entrades i sortides del programa. Tant el teclat com tots els controladors de l'FPGA són inputs directes de MAIN. MAIN distribueix els inputs entre les entitats subordinades, que al mateix temps envien els seus outputs a MAIN en forma de senyals internes. Generada la trama S/PDIF, MAIN l'envia directament al LED.

Veiem que dins del MAIN hi ha dos grans blocs: MIDI i SWG_SPDIF, el segon dels quals es divideix en els blocs SWG i SPDIF. A partir d'ara, els noms de tots els sub-programes vindran encapçalats pel bloc al que pertanyen per precisar la seva funció.

MIDI_Top (MIDI)

MIDI_Top és l'entitat controladora del bloc MIDI. Malgrat que subordinada a MAIN, MIDI_Top exerceix una funció similar: pren la informació del teclat a través de MAIN i l'envia a MIDI_Core, que s'encarregarà de traduir-la i processar-la. Un cop obtinguts els paràmetres MIDI de l'àudio a reproduir, MIDI_Top els envia al següent bloc.

En versions anteriors, MIDI_Top gestionava els subprogrames referents als diferents canals d'àudio i s'encarregava de repartir els paràmetres MIDI als canals corresponents. Tanmateix, sabent que només farem ús d'un sol canal d'àudio, aquesta funcionalitat s'ha eliminat per simplificar el programa.

MIDI_Core (MIDI)

MIDI_Core, com indica el seu nom, és el nucli del bloc MIDI. La seva funció és processar les dades que el teclat ha comunicat a l'FPGA i que la UART ha traduït, transofmant-les, mitjançant una màquina d'estats, en els paràmetres MIDI fonamentals: *note* i *velocity*.

També governa la UART, dividida en dos subprogrames, traslladant informació de l'un a l'altre (vegeu els dos subprogrames següents).

UART_Rx (MIDI)

UART_Rx és el subprograma que s'encarrega de llegir la informació en sèrie transmesa del teclat i aplegar-la en bytes de dades, que un cop complets, envia de manera cíclica al MIDI_Core.

No obstant, per si sol, UART_Rx no seria capaç de traduir correctament ni un sol bit d'informació. Necessita d'un segon subprograma que li digui quan ha de llegir i quan no.

UART_Rx_Counter (MIDI)

El subprograma UART_Rx_Counter és un generador de temps de bit que genera ticks a una freqüència 16 vegades major que la freqüència de comunicació del teclat.

Aquests ticks són enviats a la UART_Rx a través de MIDI_Core, i indiquen a la UART_Rx quan ha d'efectuar una lectura. D'acord amb el funcionament d'una UART (explicat a la secció *6.1.1 MIDI Interface*), UART_Rx gestiona cada una de les lectures de la manera més convenient per obtenir el byte de dades. UART_Rx_Counter simplement s'encarrega d'emetre els ticks, no li importa certament quan s'han d'admetre dades del teclat i quan no.

SWG_SPDIF_Top (SWG, SPDIF)

SWG_SPDIF_Top és l'homònim de MIDI_Top, però pels blocs SWG i SPDIF. El fet que SWG i SPDIF vinguin regits pel mateix programa controlador es deu a una simple qüestió de comoditat. La informació d'àudio sortint del bloc SWG és de forma directa una secció de la trama S/PDIF generada al bloc SPDIF (concretament ocupa els bits del 8 al 27 de cada trama).

Així doncs, aquesta entitat controladora rep i reparteix les senyals entrants de MAIN entre els blocs SWG i SPDIF i relaciona les senyals adients entre ambdós.

Channel freq (SWG)

Channel freq és el subprograma que s'encarrega de crear la nota. Per crear la nota entenem la modulació de la forma d'ona de l'instrument seleccionat, reproduint-la més ràpid o més lenta (escurçant-la o estirant-la) perquè soni més aguda o més greu.

Per fer-ho, Channel freq adopta el comportament d'un clock variable (que és el que farà servir Sound source posteriorment): rep del bloc MIDI el paràmetre *note* i emet polsos de rellotge a una freqüència o a una altra depenent del seu valor.

Malgrat que al principi Channel freq fos un programa curt, ja que determinava la freqüència dels ticks mitjançant una fórmula matemàtica que girava al voltant de la variable *note*, van sorgir problemes processant operacions decimals. Al final, per esquivar els problemes, mitjançant macros de l'Excel es va crear taula que determinava la freqüència dels ticks per cada valor de *note*. L'script va perdre en dimensions, però el resultat van guanyar en eficiència.

Sound source (SWG)

El Sound source és el creador del timbre i la intensitat del so.

Pren les dades dels quatre botons útils de l'FPGA a través dels que estableix quin instrument és el que ha de sonar (timbre) i a quin volum ha de sonar (intensitat).

En aquesta versió del Synthesizer, el Sound source contempla tres instruments i sis nivells de volum.

Per definir cada instrument, el Sound source conté la seva forma d'ona característica dividida en 256 punts que reprodueix cíclicament mentre estigui premuda una tecla del teclat. L'usuari pot escollir l'instrument que vol tocar a través dels botons, que el programa processa seleccionant la forma d'ona emmagatzemada al seu script.

Per definir el volum, Sound source multiplica el valor del punt actual de la forma d'ona per la variable *volume*, que l'usuari pot variar apretant els botons adients de l'FPGA. La variable *volume* pot valdre 0, 1, 2, 3, 4 o 5.

La informació d'àudio final consisteix en el registre *sample*, de 20 bits de longitud, que conté el valor del punt actual de la forma d'ona multiplicat pel *volume*.

Serialiser (SPDIF)

El Serialiser és el subprograma encarregat de crear les trames S/PDIF fent ús de la informació d'àudio final provinent del Sound source.

Com hem vist a la secció *4.3 Format de l'S/PDIF*, dins dels 64 bits que conformen cada trama, els vint bits que van del 8 al 19 són els que corresponen a la informació d'àudio. El Serialiser crea cíclicament trames S/PDIF adosant directament (en els bits del 8 al 19) la informació final d'àudio.

No obstant, per complir amb el protocol S/PDIF, el Serialiser necessita sincronitzar-se, doncs ha d'enviar trames en sèrie a 44.100 *bps*. En cas de no aconseguir una sincronització perfecta, les dades emeses a l'exterior no serien reconegudes i no es reproduiria l'àudio.

Per assolir aquesta sincronització, el Serialiser necessita el complement Timebase.

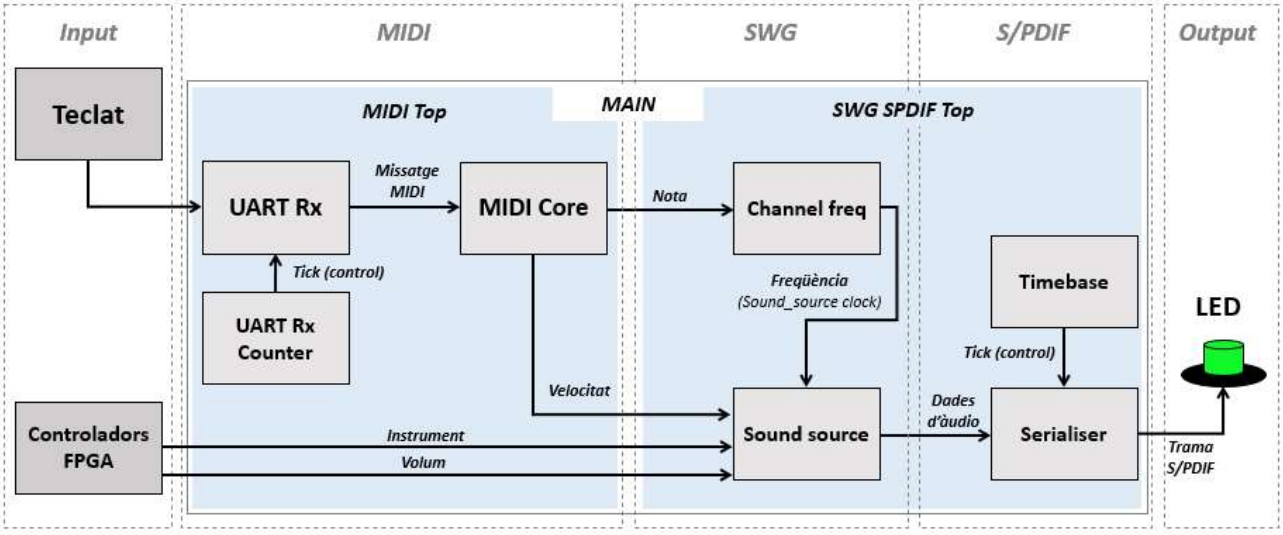
Timebase (SPDIF)

El Timebase és el darrer subprograma que considerarem. És l'encarregat de calcular la freqüència dels enviaments de les trames S/PDIF del Serialiser, que controla a base de ticks, seguint un procediment similar al que hem vist en el Channel freq del bloc SWG.

La gran diferència entre el Timebase i el Channel freq és que el Timebase necessita un nivell de precisió molt elevat per aconseguir sincronitzar l'enviament de les trames a la freqüència indicada i per això disposa d'un complex cos computacional, que contempla els residus decimals (que anomena errors) en els seus càlculs per enviar els bits de les trames de la forma més precisa possible. El resultat és una transmissió robusta de les trames S/PDIF.

Nota: Per veure amb detall cadascun dels programes explicats en aquesta secció, vegeu els annexos.

L'esquema general de Synthesizer és el següent:



Imatge 27. Esquema de Synthesizer, el programa final.

6.2.1 Millores

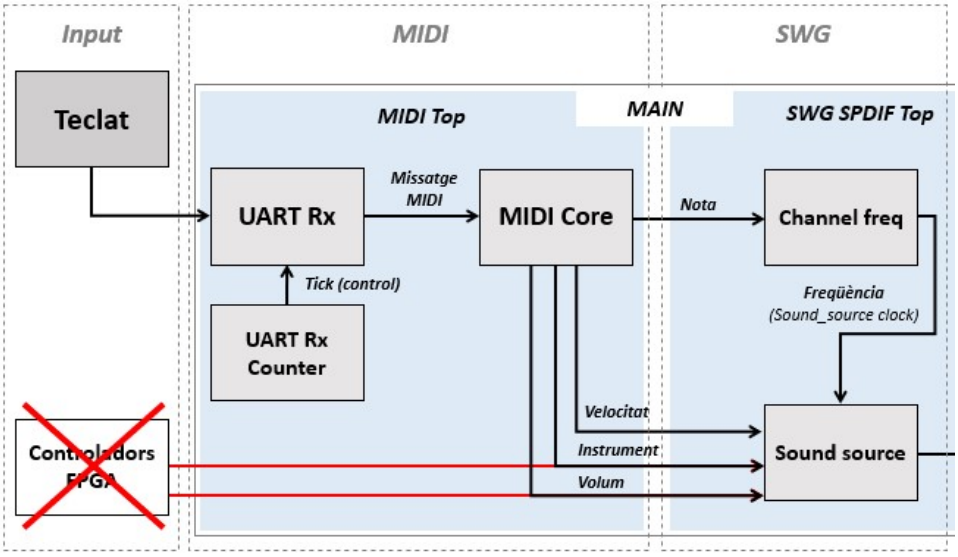
Degut a un seguit de complicacions durant el desenvolupament del projecte (entre elles, les mencionades al capítol *5. Hardware*) no es va poder seguir l'schedule proposat al principi. Això va tenir conseqüències a l'hora d'assolir totes les metes plantejades. Així doncs, en aquesta secció, es proposen com a millores futures aquestes metes no aconseguides per mera qüestió de temps.

A. Teclat com a control únic

La primera de les millores consisteix en abandonar els controls de l'FPGA per aprofitar les dades MIDI enviades pel teclat per controlar l'instrument i el volum.

Aquesta millora és relativament senzilla, doncs consisteix en modificar la màquina d'estats de MIDI_Core que converteix les dades en sèrie del teclat en missatges MIDI.

Per controlar el volum a través del teclat, podríem fer servir la rodeta. Quan es rebi els dos bytes d'*status* que fan referència al moviment de la rodeta (Control Change 0B 07), automàticament entraríem a modificar el volum; el següent byte de dades ens donarien el nou valor multiplicador de volum.



Imatge 28. Programa modificat si apliquéssim la millora A.

Per controlar l'instrument, com que el Keystation Mini 32 no disposa d'un botó específic per canviar d'instrument, podríem aprofitar el botó del controlador Sustain, que envia els

bytes d'*status* 0B 40 seguits del byte de dades 7F si està ON i 00 si està OFF. En aquest sentit, sempre que el Sustain canviés de ON a OFF i viceversa, es podria canviar d'instrument en una seqüència ordenada d'instruments.

B. Més instruments

En aquest projecte només s'han desenvolupat tres instruments diferents; ona sinusoïdal, grand piano i gaita. No obstant, aconseguint l'equació adient, mitjançant la macro de SWG es pot obtenir una forma d'ona definitòria de qualsevol instrument nou i afegir-se al programa d'una forma senzilla.

Tanmateix, un número elevat d'instruments hauria de gravar-se a la memòria interna de l'FPGA per evitar de carregar els 256 valors de cada una de les formes d'ona una vegada i una altra sempre que volguéssim inicialitzar el Synthesizer.

C. Més controls

En aquest projecte només incloem els controls del volum, la nota i l'instrument. Amb això en tenim suficient per tocar cançons, però podem anar més enllà en l'edició de sons. Com a tercera millora es podria implementar el Sustain (que manté una nota activa durant un període de temps, podent-hi tocar altres notes al damunt) o el Pitch (que modula la freqüència de la nota durant un instant de temps).

Per introduir aquests controls no només necessitaríem atribuir-los un controlador (botó, rodeta, interruptor, etc.) sinó modificar la màquina d'estats de MIDI_Core d'una forma similar a *A. Teclat com a control únic* i programar els controls al Sound_source.

Malgrat que aquesta és una millora més laboriosa, no presenta una gran complexitat.

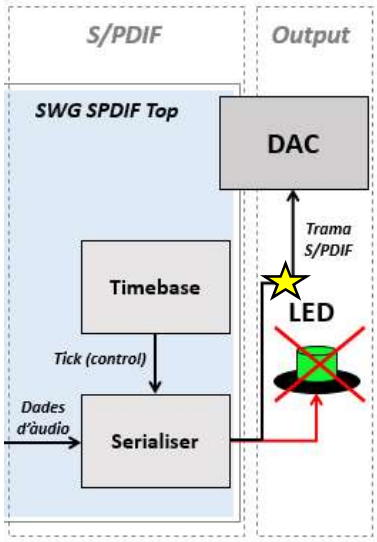
D. Connexió directa FPGA - DAC

Malgrat que emetre les trames S/PDIF via LED i fer que el DAC les llegeixi acostant l'extrem d'un cable de fibra òptica pugui semblar curiós, és certament un sistema simplificat i poc pràctic, ja que obliga a mantenir el cable damunt del LED i si aquest es desvia, la música deixa de sonar.

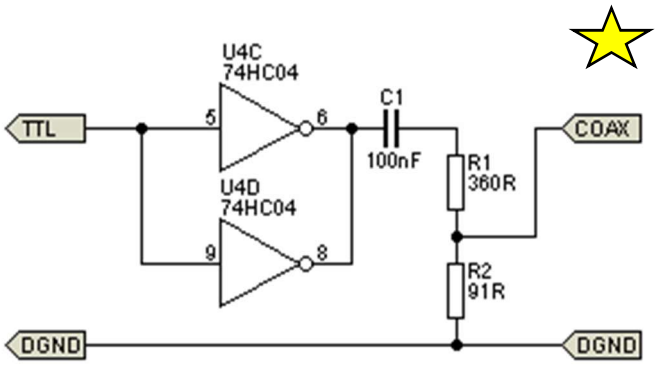
Per tal de tocar el teclat durant períodes més o menys prolongats sense preocupar-se d'apuntar el cable a la llum, seria interessant muntar un circuit que permetés una

connexió segura entre la FPGA i el DAC. Aquest circuit s'annexaria a un dels ports Pmod sobrants de la FPGA que permetria, no només emetre les dades pel pin TX sinó que l'alimentaria. Aquesta alimentació podria ser bé de 5V o 3,3V, doncs els ports Pmod de la Basts3 disposen d'ambdues.

El circuit hauria de rebre les trames S/PDIF de l'FPGA i enviar-les a un connector de cable coaxial. El cable coaxial substituiria al de fibra òptica. El DAC emprat en aquest projecte (Pozor DAC01) admet tant coaxial com fibra òptica, així que no presentaria cap problema.



Imatge 29. Programa modificat si apliquéssim la millora D.



Imatge 30. Exemple del circuit FPGA-DAC proposat.

7. CONCLUSIONS

La idea de muntar un piano digital va sorgir arran de la curiositat d'entendre com funciona un sintetitzador musical i l'àmplia llibertat que oferia la seva realització, doncs un projecte d'aquestes característiques requereix una part important de personalitat i imaginació.

No negaré que el dominar el llenguatge VHDL també hi ha jugat a favor, cosa que no podria dir del MIDI o l'S/PDIF; abans de començar el treball no en sabia gairebé res de cap dels dos.

Tanmateix, amagada a la xarxa existeix una gran comunitat aplegada al voltant del MIDI, que va ajudar-me a comprendre els fonaments d'aquest protocol mentre que amb el David apreníem a amansar-lo.

L'S/PDIF, la segona pedra al mig del camí, vaig superar-la al submergir-me en blogs oblidats, on altres aficionats havien deixat, prop de deu anys enrere, consells i programes fallits que vaig anar ajuntant i polint fins que un dia va sonar música per l'altaveu.

Dominat el VHDL, amansat el MIDI i superat l'S/PDIF, al final el piano va funcionar tan bé com s'esperava.

Si bé no he disposat de tot el temps necessari per fer d'aquest piano un aparell complet, equipat amb totes les funcions que se m'han ocorregut al llarg d'aquests mesos i amb una alimentació més pràctica per portar-lo amunt i avall, m'hauré de conformar amb donar per vàlides les millores proposades i aplicar-les al projecte més endavant.

8. BIBLIOGRAFIA

QUÈ ÉS MIDI?

1. HISPANOSONIC: *Protocolo MIDI*, 02/02/19. https://www.hispasonic.com/reportajes/protocolo-midi/13

2. LEARN SPARKFUN: *MIDI Tutorial*, 02/02/19. https://learn.sparkfun.com/tutorials/midi-tutorial/all#messages

3. WIKIPEDIA: *MIDI*, 26/05. https://es.wikipedia.org/wiki/MIDI

4. DIFFUSION MAGAZINE: *Historia del MIDI*, 26/05. http://www.diffusionmagazine.com/index.php/biblioteca/categorias/historia/335-historia-del-midi

5. FUTURE MUSIC: *Mil años de música electrónica: Telharmonium de Thaddeus Cahill*, 26/05. http://www.futuremusic-es.com/mil-anos-de-musica-electronica-telharmonium-de-thaddeus-cahill/

6. WIKIPEDIA: *Panic button*, 18/02. https://en.wikipedia.org/wiki/Panic_button#MIDI

7. SPIKENZIELABS: *Serial MIDI*, 30/05/19. http://alsitecno.com/2015/03/19/que-es-spdif-y-cuando-deberias-de-utilizarlo

QUÈ ÉS L'S/PDIF?

1. HwB: S/PDIF: *The interface*, 27/02/19. http://www.hardwarebook.info/S/PDIF#The_interface

2. ACADEMIC: *Biphase mark code*, 27/02/19. http://enacademic.com/dic.nsf/enwiki/1045852

3. ALSITECNO: *¿Qué es S/PDIF y cuando deberías usarlo?*, 04/03/19. http://alsitecno.com/2015/03/19/que-es-spdif-y-cuando-deberias-de-utilizarlo

4. WIKIPEDIA: *Protocolo S/PDIF*, 10/03/19. https://es.wikipedia.org/wiki/Protocolo_S/PDIF

PROGRAMARI

1. DAVID SOLER: *UART*, 02/02/19.

2. GITHUB: *FPGA MIDI synth*, 21/03/19. https://github.com/rene-dev/fpga-midi-synth

3. HAMSTERWORKS: *SPDIF_out*, 25/02/19. http://hamsterworks.co.nz/mediawiki/index.php/SPDIF_out

4. SOUND WHISTES: *Project85*, 18/03/19. http://sound.whsites.net/project85.htm

9. AGRAÏMENTS

Per concloure aquestes pàgines, m'agradaria agrair tot l'esforç i la paciència inacabable del David Soler, que durant els quatre mesos que va allargar-se aquest projecte va decidir sacrificar les valuoses tardes del divendres, tancat al seu despatx pel bé d'un piano que no volia funcionar. Sense cap mena de dubte, realitzar aquest treball hauria estat inviable sense la seva ajuda.

10. ANNEXOS

10.1 Synthesizer

El Synthesizer és el programa final. Rep la informació MIDI del teclat i la converteix en trames d'S/PDIF que expulsa a través d'un dels LEDs de l'FPGA.

A continuació es presenten tots els sub-programes (o entitats) que el conformen.

10.1.1 *MAIN.vhd*

```
-- Gestiona les entrades i sortides de l'FPGA i les senyals entre els dos
grans blocs del programa: el MIDI i l'SWG + SPDIF.


library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;



entity MAIN is

    port (clk               : in std_logic;

          midi_in           : in std_logic;

          sound_on_off      : in std_logic;

          vol_up, vol_down  : in std_logic;

          tone_up, tone_down: in std_logic;

          spdif_out         : out std_logic);

end MAIN;


architecture Behavioral of MAIN is


component MIDI_Top is

    port (clk     : in  std_logic;

          midi_in : in  std_logic;

          note    : out std_logic_vector (6 downto 0);

          velo    : out std_logic_vector (6 downto 0));
```

```
end component;

component SWG_SPDIF_Top is
    port ( clk              : in  STD_LOGIC;
           spdif_out        : out  STD_LOGIC;
           note             : in std_logic_vector (6 downto 0);
           velo             : in std_logic_vector (6 downto 0);
           vol_up, vol_down  : in std_logic;
           tone_up, tone_down: in std_logic;
           sw               : in std_logic);
end component;

signal Audio: std_logic;
signal note: std_logic_vector (6 downto 0);
signal velo: std_logic_vector (6 downto 0);

begin

MIDI: MIDI_Top
    port map (clk => clk,
              midi_in => midi_in,
              note => note,
              velo => velo);

SWG_SPDIF: SWG_SPDIF_Top
    port map (clk => clk,
              spdif_out => spdif_out,
              note => note,
              velo => velo,
              sw => sound_on_off,
              vol_up => vol_up,
              vol_down => vol_down,
              tone_up => tone_up,
              tone_down => tone_down);

end Behavioral;
```

10.1.2 *MIDI Top.vhd*

```
-- Gestiona les dades obtingudes pel MIDI Core i enllaça les seves senyals
internes.


library ieee;

use ieee.std_logic_1164.all;

use ieee.numeric_std.all;

library work;

use work.all;


entity MIDI_Top is port(

   clk      : in  std_logic;

   midi_in  : in  std_logic;

   note     : out std_logic_vector (6 downto 0);

   velo     : out std_logic_vector (6 downto 0)

   );

end MIDI_Top;


architecture rtl of MIDI_Top is


component MIDI_Core is port(

   clk       : in  std_logic;

   midi_in   : in  std_logic;

   midi_new  : out std_logic := '0';

   midi_ch   : out std_logic_vector(3 downto 0) := "0000";

   midi_note : out std_logic_vector(6 downto 0) := "0000000"; -- note

   midi_velo : out std_logic_vector(6 downto 0) := "0000000"  -- velocity

   );

end component;


component channel is port(

   clk      : in  std_logic;

   note_on  : in  std_logic;

   note_in  : in  std_logic_vector(6 downto 0);

   velocity : in  std_logic_vector(6 downto 0);

   volume   : in  std_logic_vector(6 downto 0);
```

```
   audio_out: out std_logic_vector(7 downto 0)
   );
end component;

signal midi_new  : std_logic := '0';
signal midi_ch   : std_logic_vector(3 downto 0);
signal midi_note : std_logic_vector(6 downto 0);
signal midi_velo : std_logic_vector(6 downto 0);

type note_on_type is array (integer range 0 to 15) of std_logic; -- Per cada
paràmetre MIDI hi ha 16 canals (com veiem al declarar l'array 0 to 15).
signal note_on : note_on_type;

type note_in_type is array (integer range 0 to 15) of std_logic_vector(6
downto 0);
signal note_in : note_in_type;

type velocity_type is array (integer range 0 to 15) of std_logic_vector(6
downto 0);
signal velocity : velocity_type;

type volume_type is array (integer range 0 to 15) of std_logic_vector(6 downto
0);
signal volume : volume_type;

type audio_type is array (integer range 0 to 15) of std_logic_vector(7 downto
0);
signal audio_ch : audio_type;

begin

MIDI_MIDICore: MIDI_Core
    port map(
       clk       => clk,
       midi_in   => midi_in,
       midi_new  => midi_new,
       midi_ch   => midi_ch,
       midi_note => midi_note,
       midi_velo => midi_velo);
```

```
note <= midi_note;
velo <= midi_velo;

process(clk,midi_in) begin
if clk'event and clk ='1' then
if(midi_new = '1') then
   note_on (to_integer(unsigned(midi_ch))) <= '1';
   note_in (to_integer(unsigned(midi_ch))) <= midi_note;
   velocity(to_integer(unsigned(midi_ch))) <= midi_velo;
   volume  (to_integer(unsigned(midi_ch))) <= "0010000"; -- Volum per defecte.
   else
   note_on (to_integer(unsigned(midi_ch))) <= '0';
end if;
end if;
end process;

end rtl;
```

10.1.3 *MIDI Core.vhd*

```
-- Mitjançant una màquina d'estats i les dades obtingudes a través de la UART,
defineix els bytes de note i velocity que definiran la nota i si aquesta ha de
sonar o no.
-- Gestiona les senyals internes entre UART Rx i UART Rx Counter.

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.all;

entity MIDI_Core is port(
   clk       : in  std_logic;
   midi_in   : in  std_logic;
```

```
   midi_new  : out std_logic := '0';
   midi_ch   : out std_logic_vector(3 downto 0) := "0000";
   midi_note : out std_logic_vector(6 downto 0) := "0000000"; -- note
   midi_velo : out std_logic_vector(6 downto 0) := "0000000" -- velocity
   );
end MIDI_Core;

architecture rtl of MIDI_Core is

type midi_state_type is (status,note_off,note_on,velocity);
signal uart_busy            : std_logic;
signal midi_data            : std_logic_vector(7 downto 0);
signal midi_state           : midi_state_type := status;
signal next_midi_state      : midi_state_type := status;
signal tick, falling, off   : std_logic := '0';

component UART_Rx is
       generic (dbit      : integer := 8;
                  sb_tick   : integer:= 16);
       port (clk          : in std_logic;
            reset         : in std_logic;
            rx, s_tick    : in std_logic;
              rx_done_tick  : out std_logic;
              dout          : out std_logic_vector (7 downto 0));
end component;

component UART_Rx_Counter is
       generic (n    : integer := 8;
                  m : integer := 200); --100MHz (clk) = (31250 bps * 16) * 200
       port ( clk     : in std_logic;
              reset   : in std_logic;
              max_tick: out std_logic);
end component;

begin

MIDI_UART: UART_Rx
```

```
       generic map (dbit => 8, sb_tick => 16)
    port map (clk => clk,
              reset => '0',
              rx => midi_in,
              s_tick => tick,
              rx_done_tick => falling,
              dout => midi_data);                        -- clk

MIDI_UART_Counter: UART_Rx_Counter
       generic map (n => 8, m => 200)
    port map (clk => clk,
              reset => '0',
              max_tick => tick);

process begin
   wait until rising_edge(clk);
   midi_new <= '0';
   if(falling = '1') then -- Si s'ha acabat de rebre un byte MIDI...
      if(not (midi_data = "11111110" or midi_data = "11111000")) then --
active sense or clock
         case midi_state is
            when status =>
               --midi_new <= '0';
               if(midi_data(7 downto 4) = "1000") then    --note off
                  midi_state <= note_off;
                  midi_ch    <= midi_data(3 downto 0);
                  midi_velo  <= "0000000";
                  off <= '1';
               elsif(midi_data(7 downto 4) = "1001") then --note on
                  midi_state <= note_on;
                  midi_ch    <= midi_data(3 downto 0);
                  off <= '0';
               end if;
            when note_on =>
               if(midi_data(7) = '0') then
                  midi_note  <= midi_data(6 downto 0);
                  midi_state <= velocity;
```

```
                else
                    midi_state <= status;
                end if;
            when note_off =>
                if(midi_data(7) = '0') then
                    midi_note  <= midi_data(6 downto 0);
                    midi_state <= velocity;
                else
                    midi_state <= status;
                end if;
            when velocity =>
                if(midi_data(7) = '0') then
                    if(off = '0') then
                    midi_velo <= midi_data(6 downto 0);
                    else
                    midi_velo <= "0000000";
                    end if;
                    midi_state <= status;
                    midi_new   <= '1';
                else
                    midi_state <= status;
                end if;
         end case;
      end if;
   end if;
end process;
end rtl;
```

10.1.4 *UART Rx.vhd*

```
-- Rep les dades en sèrie del teclat. S'ajuda de UART Rx Counter per definir
quan ha de llegir i quan no per fer la lectura correcta.


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
use IEEE.NUMERIC_STD.ALL;

entity UART_Rx is
	generic (
		dbit		: integer := 8;
		sb_tick		: integer:= 16);
	port (
		clk		: in std_logic;
		reset		: in std_logic;
		rx		: in std_logic;
		s_tick		: in std_logic;
		rx_done_tick	: out std_logic;
		dout		: out std_logic_vector (7 downto 0));
end UART_Rx;

architecture Behavioral of UART_Rx is

	type state_type is (idle, start, data, stop);
	signal state_reg, state_next	: state_type;
	signal s_reg, s_next		: unsigned (3 downto 0);
	signal n_reg, n_next		: unsigned (2 downto 0);
	signal b_reg, b_next		: std_logic_vector (7 downto 0);

begin
-- FSMD state & data registers
process (clk, reset) begin
	if reset = '1' then
		state_reg <= idle;
		s_reg <= (others => '0');
		n_reg <= (others => '0');
		b_reg <= (others => '0');
	elsif (clk'event and clk ='1') then
		state_reg <= state_next;
		s_reg <= s_next;
		n_reg <= n_next;
		b_reg <= b_next;
	end if;
```

```
end process;

-- next-state logic & data path functional units/routing
process (state_reg, s_reg, n_reg, b_reg, s_tick, rx) begin
	state_next <= state_reg;
	s_next <= s_reg;
	n_next <= n_reg;
	b_next <= b_reg;
	rx_done_tick <= '0';
	case state_reg is
		when idle =>
			if rx = '0' then
				state_next <= start;
				s_next <= (others => '0');
			end if;
		when start =>
			if s_tick = '1' then
				if s_reg = 7 then
					state_next <= data;
					s_next <= (others => '0');
					n_next <= (others => '0');
				else
					s_next <= s_reg + 1;
				end if;
			end if;
		when data =>
			if s_tick = '1' then
				if s_reg = 15 then
					s_next <= (others => '0');
					b_next <= rx & b_reg (7 downto 1);
					if n_reg = dbit -1 then
						state_next <= stop;
					else
						n_next <= n_reg+1;
					end if;
				else
					s_next <= s_reg +1;
```

```
                        end if;
                    end if;
                when stop =>
                    if s_tick = '1' then
                        if s_reg = sb_tick - 1 then
                            state_next <= idle;
                            rx_done_tick <= '1';
                        else
                            s_next <= s_reg +1;
                        end if;
                    end if;
        end case;
end process;

dout <= b_reg;

end Behavioral;
```

10.1.5 *UART Rx Counter.vhd*

```
-- Generador de temps de bit de UART Rx.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity UART_Rx_Counter is
    generic ( n:integer := 8;
              m : integer := 200);
    port ( clk     : in std_logic;
           reset    : in std_logic;
           max_tick : out std_logic);
end UART_Rx_Counter;

architecture Behavioral of UART_Rx_Counter is
```

```
       signal r_reg  : unsigned (n-1 downto 0);
       signal r_next : unsigned (n-1 downto 0);

begin

process ( clk,reset) begin
       if reset = '1' then
              r_reg <= (others => '0');
       elsif clk'event and clk = '1' then
              r_reg <= r_next;
       end if;
end process;

r_next <= (others => '0') when r_reg = m-1 else r_reg +1;

max_tick <= '1' when r_reg = m-1 else '0';

end Behavioral;
```

10.1.6 *SWG SPDIF Top.vhd*

```
-- Governa els inputs i els outputs de l'SWG i l'S/PDIF així com assegura la
transmissió de les dades d'àudio sortints del bloc SWG al S/PDIF.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity SWG_SPDIF_Top is
    Port ( clk                  : in  std_logic;
           note,velo            : in std_logic_vector (6 downto 0);
           sw                   : in std_logic;
           vol_up, vol_down     : in std_logic;
           tone_up, tone_down   : in std_logic;
           spdif_out            : out  std_logic);
end SWG_SPDIF_Top;
```

```
architecture Behavioral of SWG_SPDIF_Top is

   COMPONENT serialiser
   PORT(
      clk          : in std_logic;
      auxAudioBits : in std_logic_vector(3 downto 0);
      sample       : in std_logic_vector(19 downto 0);
      nextSample   : out std_logic;
      channelA     : out std_logic;
      spdifOut     : out std_logic
      );
   END COMPONENT;

   component Channel_freq is
       Port(
          clk      : in  std_logic;
          note     : in  std_logic_vector (6 downto 0);
          ch_clk   : out  std_logic
       );
   end component;

   component Sound_source is
       Port(
          clk               : in  std_logic;
          ch_clk            : in  std_logic;
          velo              : in std_logic_vector (6 downto 0);
          channelA          : in std_logic;
          next_sample       : in std_logic;
          vol_up, vol_down  : in std_logic;
          tone_up, tone_down: in std_logic;
          sw                : in std_logic;
          sample            : out std_logic_vector (19 downto 0)
       );
   end component;

   signal nextSample    : std_logic;
```

```
   signal channelA      : std_logic;
   signal sample        : std_logic_vector(19 downto 0);
   signal ch_clk        : std_logic;
begin

    SPDIF_Serialiser: serialiser
        PORT MAP(
          clk          => clk,
          auxAudioBits => "0000",
          sample       => sample,
          nextSample   => nextSample,
          channelA     => channelA,
          spdifOut     => spdif_out
        );

    SWG_Channel_freq: Channel_freq
        port map (
            clk => clk,
            note => note,
            ch_clk => ch_clk
        );

    SWG_Sound_source: Sound_source
        port map (
            clk => clk,
            ch_clk => ch_clk,
            sample => sample,
            next_sample => nextSample,
            vol_up => vol_up,
            vol_down => vol_down,
            channelA => channelA,
            tone_up => tone_up,
            velo => velo,
            tone_down => tone_down,
            sw => sw
         );
```

```
end Behavioral;
```

10.1.7 *Serialiser.vhd*

```
-- Converteix les dades d'àudio emeses per l'SWG en trames S/PDIF.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Serialiser is
    Port ( clk             : in  STD_LOGIC;
           auxAudioBits    : in  STD_LOGIC_VECTOR (3 downto 0);
           sample          : in  STD_LOGIC_VECTOR (19 downto 0);
           nextSample      : out STD_LOGIC;
           channelA        : out STD_LOGIC;
           spdifOut        : out STD_LOGIC);
end Serialiser;

architecture Behavioral of Serialiser is
   COMPONENT Timebase
   PORT(
      clk            : IN std_logic;
      bitclock       : OUT std_logic;
      loadSerialiser : OUT std_logic
      );
   END COMPONENT;

   signal bitclock      : std_logic;
   signal loadSerialiser: std_logic;

   signal bits          : std_logic_vector(63 downto 0) := (others => '0'); --
Bits sencers de la trama.
   signal current       : std_logic := '0';
   signal preamble      : STD_LOGIC_VECTOR (7 downto 0);
   signal sample2       : STD_LOGIC_VECTOR (19 downto 0);
   signal subframeCount : STD_LOGIC_VECTOR (7 downto 0) := "00000000";
   signal parity        : STD_LOGIC;
```

```
  constant validity     : STD_LOGIC := '0';
   constant subcode     : STD_LOGIC := '0';
   constant channelStatus: STD_LOGIC := '0';


begin
   SPDIF_Timebase: Timebase PORT MAP(
      clk => clk,
      bitclock => bitclock,
      loadSerialiser => loadSerialiser
   );


   sample2     <= sample(19 downto 4) & "0000";
   spdifOut    <= current;
   nextSample  <= loadSerialiser;
   channelA    <= subFrameCount(0);


   -- Càlcul del bit de paritat:
   parity <= auxAudioBits(3) xor auxAudioBits(2) xor auxAudioBits(1) xor
auxAudioBits(0) xor
             sample2(19)      xor sample2(18)      xor sample2(17)      xor
sample2(16)      xor
             sample2(15)      xor sample2(14)      xor sample2(13)      xor
sample2(12)      xor
             sample2(11)      xor sample2(10)      xor sample2(9)       xor
sample2(8)       xor
             sample2(7)       xor sample2(6)       xor sample2(5)       xor
sample2(4)       xor
             sample2(3)       xor sample2(2)       xor sample2(1)       xor
sample2(0)       xor
             subcode         xor validity        xor channelStatus   xor '0';


   process (subFrameCount)
   begin
      if subframeCount = "00000000" then
         preamble <= "00111001";
      else
         if subframeCount(0) = '0' then
            preamble <= "11001001";
         else
```

```
            preamble <= "01101001";
         end if;
      end if;
   end process;


   process(bits, clk, bitclock, loadSerialiser, preamble, auxAudioBits,
sample, parity)
   begin
      if clk'event and clk = '1' then
         if loadSerialiser = '1' then
            bits <= parity    & "1" & channelStatus    & "1" & subcode
& "1" & validity         & "1" &
               sample2(19)      & "1" & sample2(18)      & "1" & sample2(17)
& "1" & sample2(16)      & "1" &
               sample2(15)      & "1" & sample2(14)      & "1" & sample2(13)
& "1" & sample2(12)      & "1" &
               sample2(11)      & "1" & sample2(10)      & "1" & sample2( 9)
& "1" & sample2( 8)      & "1" &
               sample2( 7)      & "1" & sample2( 6)      & "1" & sample2( 5)
& "1" & sample2( 4)      & "1" &
               sample2( 3)      & "1" & sample2( 2)      & "1" & sample2( 1)
& "1" & sample2( 0)      & "1" &
               auxAudioBits(3)& "1" & auxAudioBits(2) & "1" & auxAudioBits(1)
& "1" & auxAudioBits(0) & "1" &
               preamble;


            if subframeCount = 191 then
               subFrameCount <= (others => '0');
            else
               subFrameCount <= subFrameCount +1;
            end if;


         elsif  bitclock = '1' then
            current <= current xor bits(0) xor '0';
            bits <= "0" & bits(63 downto 1);
         end if;
      end if;
   end process;
end Behavioral;
```

10.1.9 *Timebase.vhd*

```
-- Genera la senyal de clk per a l'output S/PDIF.


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity Timebase is
    Port ( clk : in  STD_LOGIC;
           bitclock : out  STD_LOGIC;
           loadSerialiser : OUT std_logic);
end Timebase;



architecture Behavioral of Timebase is
   type reg is record
       state            : std_logic_vector(4 downto 0);
       errorTotal       : std_logic_vector(9 downto 0);
        bitCount         : std_logic_vector(5 downto 0);
       bitClock         : std_logic;
          loadSerialiser   : std_logic;
   end record;


   signal r : reg := ((others => '0'), (others => '0'), (others => '0'), '0',
'0');
   signal n : reg;


   constant terminalCount : natural := 882;
   constant errorStep      : natural := 631;
begin
    loadSerialiser <= r.loadSerialiser;
    bitClock        <= r.bitClock;


   process(clk,r)
   begin
      n <= r;
```

```
      n.bitclock                 <= '0';
      n.loadSerialiser           <= '0';
      n.state                    <= r.state+1;
      case r.state is
         when "00000" =>
            n.bitclock <= '1';
         when "00001" =>
                    n.bitcount <= r.bitcount + 1;
         when "00010" =>
                    if n.bitcount = "000000" then
                           n.loadSerialiser <= '1';
                    end if;
         when "10000" =>
            if r.errorTotal < terminalCount - errorStep then
               n.state <= "00000";
               n.errorTotal <= r.errorTotal + errorStep;
            else
               n.errorTotal <= r.errorTotal + errorStep - terminalCount;
            end if;
         when "10001" =>
            n.state <= "00000";
         when others =>
            n.state <= r.state+1;
      end case;
   end process;

   process(clk, n)
   begin
      if clk'event and clk = '1' then
         r <= n;
      end if;
   end process;
end Behavioral;
```

10.1.10 *Channel freq.vhd*

```
-- Estableix la freqüència de reproducció del so en funció de la tecla premuda
del teclat. Això estira o contrau la forma d'ona a reproduir, creant-se una
nota o una altra.


library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;

use IEEE.NUMERIC_STD.ALL;


entity Channel_freq is

    Port ( clk      : in STD_LOGIC;

           note     : in STD_LOGIC_VECTOR (6 downto 0);

           ch_clk   : out STD_LOGIC);

end Channel_freq;


architecture Behavioral of Channel_freq is


signal freq         : integer;

signal current_freq : integer range 82 to 125439:=82;

signal cnt_clk      : integer:=0;

signal note_val     : integer:=0;

signal note_new     : std_logic;

signal current_note : std_logic_vector (6 downto 0);


begin


Set_frequency: process (note)

begin

    case note is

       when "0000000" => freq <= 82;

       when "0000001" => freq <= 87;

       when "0000010" => freq <= 92;

       when "0000011" => freq <= 97;

       when "0000100" => freq <= 103;

       when "0000101" => freq <= 109;
```

```
when "0000110" => freq <= 116;
when "0000111" => freq <= 122;
when "0001000" => freq <= 130;
when "0001001" => freq <= 137;
when "0001010" => freq <= 146;
when "0001011" => freq <= 154;
when "0001100" => freq <= 163;
when "0001101" => freq <= 173;
when "0001110" => freq <= 184;
when "0001111" => freq <= 194;
when "0010000" => freq <= 206;
when "0010001" => freq <= 218;
when "0010010" => freq <= 231;
when "0010011" => freq <= 245;
when "0010100" => freq <= 260;
when "0010101" => freq <= 275;
when "0010110" => freq <= 291;
when "0010111" => freq <= 309; --23
when "0011000" => freq <= 327;
when "0011001" => freq <= 347;
when "0011010" => freq <= 367;
when "0011011" => freq <= 389;
when "0011100" => freq <= 412;
when "0011101" => freq <= 437;
when "0011110" => freq <= 462;
when "0011111" => freq <= 490;
when "0100000" => freq <= 519;
when "0100001" => freq <= 550;
when "0100010" => freq <= 583;
when "0100011" => freq <= 617;
when "0100100" => freq <= 654;
when "0100101" => freq <= 693;
when "0100110" => freq <= 734;
when "0100111" => freq <= 778;
when "0101000" => freq <= 824;
when "0101001" => freq <= 873;
when "0101010" => freq <= 925;
```

```
when "0101011" => freq <= 980;
when "0101100" => freq <= 1038;
when "0101101" => freq <= 1100;
when "0101110" => freq <= 1166;
when "0101111" => freq <= 1235; --47
when "0110000" => freq <= 1308;
when "0110001" => freq <= 1386;
when "0110010" => freq <= 1468;
when "0110011" => freq <= 1556;
when "0110100" => freq <= 1648;
when "0110101" => freq <= 1746;
when "0110110" => freq <= 1850;
when "0110111" => freq <= 1960;
when "0111000" => freq <= 2076;
when "0111001" => freq <= 2200;
when "0111010" => freq <= 2330;
when "0111011" => freq <= 2469;
when "0111100" => freq <= 2616;
when "0111101" => freq <= 2772;
when "0111110" => freq <= 2937;
when "0111111" => freq <= 3111;
when "1000000" => freq <= 3296;
when "1000001" => freq <= 3492;
when "1000010" => freq <= 3700;
when "1000011" => freq <= 3920;
when "1000100" => freq <= 4153;
when "1000101" => freq <= 4400;
when "1000110" => freq <= 4662;
when "1000111" => freq <= 4939; --71
when "1001000" => freq <= 5233;
when "1001001" => freq <= 5544;
when "1001010" => freq <= 5873;
when "1001011" => freq <= 6223;
when "1001100" => freq <= 6593;
when "1001101" => freq <= 6985;
when "1001110" => freq <= 7400;
when "1001111" => freq <= 7840;
```

```
when "1010000" => freq <= 8306;
when "1010001" => freq <= 8800;
when "1010010" => freq <= 9323;
when "1010011" => freq <= 9878;
when "1010100" => freq <= 10465;
when "1010101" => freq <= 11087;
when "1010110" => freq <= 11747;
when "1010111" => freq <= 12445;
when "1011000" => freq <= 13185;
when "1011001" => freq <= 13969;
when "1011010" => freq <= 14800;
when "1011011" => freq <= 15680;
when "1011100" => freq <= 16612;
when "1011101" => freq <= 17600;
when "1011110" => freq <= 18647;
when "1011111" => freq <= 19755; --95
when "1100000" => freq <= 20930;
when "1100001" => freq <= 22175;
when "1100010" => freq <= 23493;
when "1100011" => freq <= 24890;
when "1100100" => freq <= 26370;
when "1100101" => freq <= 27938;
when "1100110" => freq <= 29600;
when "1100111" => freq <= 31360;
when "1101000" => freq <= 33224;
when "1101001" => freq <= 35200;
when "1101010" => freq <= 37293;
when "1101011" => freq <= 39511;
when "1101100" => freq <= 41860;
when "1101101" => freq <= 44349;
when "1101110" => freq <= 46986;
when "1101111" => freq <= 49780;
when "1110000" => freq <= 52740;
when "1110001" => freq <= 55877;
when "1110010" => freq <= 59199;
when "1110011" => freq <= 62719;
when "1110100" => freq <= 66449;
```

```
        when "1110101" => freq <= 70400;
        when "1110110" => freq <= 74586;
        when "1110111" => freq <= 79021; --119
        when "1111000" => freq <= 83720;
        when "1111001" => freq <= 88698;
        when "1111010" => freq <= 93973;
        when "1111011" => freq <= 99561;
        when "1111100" => freq <= 105481;
        when "1111101" => freq <= 111753;
        when "1111110" => freq <= 118398;
        when others    => freq <= 125439;
    end case;
end process;

detector_flanc: process (clk, note) begin
if clk'event and clk='1' then
    note_new <= '0';
    if current_note /= note then
        note_new <= '1';
        current_note <= note;
    end if;
end if;
end process;

canvi_freq: process (clk) begin
if clk'event and clk='1' then
    if note_new = '1' then
    current_freq<=freq;
    end if;
end if;
end process;

creacio_nou_clk: process(clk, note_new) begin
if clk'event and clk='1' then
    if (cnt_clk = 100_000_000/(51*current_freq)) or note_new='1' then
        cnt_clk <= 0;
        ch_clk <= '1';
```

```
    else
        cnt_clk <= cnt_clk +1;
        ch_clk <= '0';
    end if;
end if;
end process;

end Behavioral;
```

10.1.11 *Sound source.vhd*

```
-- Emmagatzema les formes d'ones de cada instrument.
-- Selecciona la forma d'ona a reproduir en funció de l'instrument escollit a
través dels botons de l'FPGA i la reprodueix mentre una tecla del teclat
estigui premuda.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity Sound_source is
    Port ( clk                  : in  STD_LOGIC;
           ch_clk               : in  STD_LOGIC;
           channelA             : in std_logic;
           velo                 : in std_logic_vector (6 downto 0);
           next_sample          : in std_logic;
           vol_up, vol_down     : in std_logic;
           tone_up, tone_down   : in std_logic;
           sw                   : in std_logic; -- sound on/off
           sample               : out  STD_LOGIC_VECTOR (19 downto 0));
end Sound_source;

architecture Behavioral of Sound_source is

   signal sampleCounter : STD_LOGIC_VECTOR(7 downto 0):= (others => '0');
```

```
   signal sample_tone   : std_logic_vector(19 downto 0);
   signal volume        : integer range 0 to 5:= 2;
   signal tone          : integer range 0 to 2:=0;
   signal q0, q1        : std_logic;
   signal vol_up_tick, vol_down_tick    : std_logic;
   signal tone_up_tick, tone_down_tick  : std_logic;
   signal value         : integer;
   signal pols_actiu    : std_logic;

begin

process(clk, ch_clk, sampleCounter)
   begin
      if clk'event and clk = '1' then
          if ch_clk= '1' then
              sampleCounter <= sampleCounter+1;
         end if;
      end if;
   end process;

 process (vol_up, vol_down, tone_up, tone_down) begin
   if (vol_up or vol_down or tone_up or tone_down)='1' then
       pols_actiu <= '1';
   else
       pols_actiu <= '0';
   end if;
 end process;

flank_detector: process (clk, pols_actiu) begin
     if clk'event and clk='1' then
         if pols_actiu='1' then
              q0<= '1';
              q1<=q0;
         else
             q0<='0';
             q1<=q0;
         end if;
```

```
     end if;
 end process;

vol_up_tick <= vol_up and q0 and not q1;
vol_down_tick <= vol_down and q0 and not q1;
tone_up_tick <= tone_up and q0 and not q1;
tone_down_tick <= tone_down and q0 and not q1;

process (clk,vol_down_tick,vol_up_tick) begin
    if clk'event and clk='1' then
        if vol_down_tick = '1' and vol_up_tick = '0' then
            if volume > 0 then
                volume <= volume -1;
            elsif volume = 0 then
                volume <= 5;
            end if;
        elsif vol_up_tick = '1' and vol_down_tick = '0' then
            if volume < 5 then
                volume <= volume +1;
            elsif volume = 5 then
                volume <= 0;
            end if;
        end if;
    end if;
end process;

process (clk,tone_down_tick,tone_up_tick) begin
    if clk'event and clk='1' then
        if tone_down_tick = '1' and tone_up_tick = '0' then
            if tone > 0 then
                tone <= tone -1;
            elsif tone = 0 then
                tone <= 2;
            end if;
        elsif tone_up_tick = '1' and tone_down_tick = '0' then
            if tone < 2 then
                tone <= tone +1;
```

```
            elsif tone = 2 then
                tone <= 0;
            end if;
        end if;
    end if;
end process;

process(sampleCounter,sw)   -- Ona sinusoïdal (valors)
begin
if sw = '1' and tone = 0 then
    case sampleCounter is
    when "00000001" => sample_tone <= x"00000";
    when "00000010" => sample_tone <= x"00506";
    when "00000011" => sample_tone <= x"00A0C";
    when "00000100" => sample_tone <= x"00F10";
    when "00000101" => sample_tone <= x"01412";
    when "00000110" => sample_tone <= x"01911";
    when "00000111" => sample_tone <= x"01E0C";
    when "00001000" => sample_tone <= x"02303";
    when "00001001" => sample_tone <= x"027F4";
    when "00001010" => sample_tone <= x"02CDF";
    when "00001011" => sample_tone <= x"031C3";
    when "00001100" => sample_tone <= x"0369F";
    when "00001101" => sample_tone <= x"03B73";
    when "00001110" => sample_tone <= x"0403D";
    when "00001111" => sample_tone <= x"044FE";
    when "00010000" => sample_tone <= x"049B4";
    when "00010001" => sample_tone <= x"04E5F";
    when "00010010" => sample_tone <= x"052FE";
    when "00010011" => sample_tone <= x"05790";
    when "00010100" => sample_tone <= x"05C14";
    when "00010101" => sample_tone <= x"0608A";
    when "00010110" => sample_tone <= x"064F1";
    when "00010111" => sample_tone <= x"06949";
    when "00011000" => sample_tone <= x"06D91";
    when "00011001" => sample_tone <= x"071C7";
    when "00011010" => sample_tone <= x"075EC";
```

```
when "00011011" => sample_tone <= x"079FF";
when "00011100" => sample_tone <= x"07DFF";
when "00011101" => sample_tone <= x"081EC";
when "00011110" => sample_tone <= x"085C4";
when "00011111" => sample_tone <= x"08988";
when "00100000" => sample_tone <= x"08D37";
when "00100001" => sample_tone <= x"090D0";
when "00100010" => sample_tone <= x"09453";
when "00100011" => sample_tone <= x"097BF";
when "00100100" => sample_tone <= x"09B13";
when "00100101" => sample_tone <= x"09E4F";
when "00100110" => sample_tone <= x"0A173";
when "00100111" => sample_tone <= x"0A47F";
when "00101000" => sample_tone <= x"0A770";
when "00101001" => sample_tone <= x"0AA48";
when "00101010" => sample_tone <= x"0AD06";
when "00101011" => sample_tone <= x"0AFA9";
when "00101100" => sample_tone <= x"0B231";
when "00101101" => sample_tone <= x"0B49D";
when "00101110" => sample_tone <= x"0B6EE";
when "00101111" => sample_tone <= x"0B922";
when "00110000" => sample_tone <= x"0BB3A";
when "00110001" => sample_tone <= x"0BD35";
when "00110010" => sample_tone <= x"0BF13";
when "00110011" => sample_tone <= x"0C0D3";
when "00110100" => sample_tone <= x"0C276";
when "00110101" => sample_tone <= x"0C3FA";
when "00110110" => sample_tone <= x"0C561";
when "00110111" => sample_tone <= x"0C6A9";
when "00111000" => sample_tone <= x"0C7D2";
when "00111001" => sample_tone <= x"0C8DD";
when "00111010" => sample_tone <= x"0C9C8";
when "00111011" => sample_tone <= x"0CA95";
when "00111100" => sample_tone <= x"0CB42";
when "00111101" => sample_tone <= x"0CBD0";
when "00111110" => sample_tone <= x"0CC3E";
when "00111111" => sample_tone <= x"0CC8D";
```

```
when "01000000" => sample_tone <= x"0CCBC";
when "01000001" => sample_tone <= x"0CCCC";
when "01000010" => sample_tone <= x"0CCBC";
when "01000011" => sample_tone <= x"0CC8D";
when "01000100" => sample_tone <= x"0CC3E";
when "01000101" => sample_tone <= x"0CBD0";
when "01000110" => sample_tone <= x"0CB42";
when "01000111" => sample_tone <= x"0CA95";
when "01001000" => sample_tone <= x"0C9C8";
when "01001001" => sample_tone <= x"0C8DD";
when "01001010" => sample_tone <= x"0C7D2";
when "01001011" => sample_tone <= x"0C6A9";
when "01001100" => sample_tone <= x"0C561";
when "01001101" => sample_tone <= x"0C3FA";
when "01001110" => sample_tone <= x"0C276";
when "01001111" => sample_tone <= x"0C0D3";
when "01010000" => sample_tone <= x"0BF13";
when "01010001" => sample_tone <= x"0BD35";
when "01010010" => sample_tone <= x"0BB3A";
when "01010011" => sample_tone <= x"0B922";
when "01010100" => sample_tone <= x"0B6EE";
when "01010101" => sample_tone <= x"0B49D";
when "01010110" => sample_tone <= x"0B231";
when "01010111" => sample_tone <= x"0AFA9";
when "01011000" => sample_tone <= x"0AD06";
when "01011001" => sample_tone <= x"0AA48";
when "01011010" => sample_tone <= x"0A770";
when "01011011" => sample_tone <= x"0A47E";
when "01011100" => sample_tone <= x"0A173";
when "01011101" => sample_tone <= x"09E4F";
when "01011110" => sample_tone <= x"09B13";
when "01011111" => sample_tone <= x"097BE";
when "01100000" => sample_tone <= x"09452";
when "01100001" => sample_tone <= x"090D0";
when "01100010" => sample_tone <= x"08D37";
when "01100011" => sample_tone <= x"08988";
when "01100100" => sample_tone <= x"085C4";
```

```
when "01100101" => sample_tone <= x"081EC";
when "01100110" => sample_tone <= x"07DFF";
when "01100111" => sample_tone <= x"079FF";
when "01101000" => sample_tone <= x"075EC";
when "01101001" => sample_tone <= x"071C7";
when "01101010" => sample_tone <= x"06D90";
when "01101011" => sample_tone <= x"06949";
when "01101100" => sample_tone <= x"064F1";
when "01101101" => sample_tone <= x"0608A";
when "01101110" => sample_tone <= x"05C14";
when "01101111" => sample_tone <= x"0578F";
when "01110000" => sample_tone <= x"052FD";
when "01110001" => sample_tone <= x"04E5F";
when "01110010" => sample_tone <= x"049B4";
when "01110011" => sample_tone <= x"044FE";
when "01110100" => sample_tone <= x"0403D";
when "01110101" => sample_tone <= x"03B72";
when "01110110" => sample_tone <= x"0369F";
when "01110111" => sample_tone <= x"031C2";
when "01111000" => sample_tone <= x"02CDE";
when "01111001" => sample_tone <= x"027F3";
when "01111010" => sample_tone <= x"02302";
when "01111011" => sample_tone <= x"01E0C";
when "01111100" => sample_tone <= x"01911";
when "01111101" => sample_tone <= x"01412";
when "01111110" => sample_tone <= x"00F10";
when "01111111" => sample_tone <= x"00A0C";
when "10000000" => sample_tone <= x"00506";
when "10000001" => sample_tone <= x"19998";
when "10000010" => sample_tone <= x"19491";
when "10000011" => sample_tone <= x"18F8C";
when "10000100" => sample_tone <= x"18A87";
when "10000101" => sample_tone <= x"18585";
when "10000110" => sample_tone <= x"18086";
when "10000111" => sample_tone <= x"17B8B";
when "10001000" => sample_tone <= x"17695";
when "10001001" => sample_tone <= x"171A4";
```

```
when "10001010" => sample_tone <= x"16CB9";
when "10001011" => sample_tone <= x"167D5";
when "10001100" => sample_tone <= x"162F9";
when "10001101" => sample_tone <= x"15E25";
when "10001110" => sample_tone <= x"1595A";
when "10001111" => sample_tone <= x"15499";
when "10010000" => sample_tone <= x"14FE3";
when "10010001" => sample_tone <= x"14B39";
when "10010010" => sample_tone <= x"1469A";
when "10010011" => sample_tone <= x"14208";
when "10010100" => sample_tone <= x"13D84";
when "10010101" => sample_tone <= x"1390D";
when "10010110" => sample_tone <= x"134A6";
when "10010111" => sample_tone <= x"1304E";
when "10011000" => sample_tone <= x"12C07";
when "10011001" => sample_tone <= x"127D0";
when "10011010" => sample_tone <= x"123AB";
when "10011011" => sample_tone <= x"11F99";
when "10011100" => sample_tone <= x"11B98";
when "10011101" => sample_tone <= x"117AC";
when "10011110" => sample_tone <= x"113D3";
when "10011111" => sample_tone <= x"1100F";
when "10100000" => sample_tone <= x"10C61";
when "10100001" => sample_tone <= x"108C8";
when "10100010" => sample_tone <= x"10545";
when "10100011" => sample_tone <= x"101D9";
when "10100100" => sample_tone <= x"0FE85";
when "10100101" => sample_tone <= x"0FB48";
when "10100110" => sample_tone <= x"0F824";
when "10100111" => sample_tone <= x"0F519";
when "10101000" => sample_tone <= x"0F227";
when "10101001" => sample_tone <= x"0EF50";
when "10101010" => sample_tone <= x"0EC92";
when "10101011" => sample_tone <= x"0E9EF";
when "10101100" => sample_tone <= x"0E767";
when "10101101" => sample_tone <= x"0E4FA";
when "10101110" => sample_tone <= x"0E2AA";
```

```
when "10101111" => sample_tone <= x"0E075";
when "10110000" => sample_tone <= x"0DE5E";
when "10110001" => sample_tone <= x"0DC63";
when "10110010" => sample_tone <= x"0DA85";
when "10110011" => sample_tone <= x"0D8C5";
when "10110100" => sample_tone <= x"0D722";
when "10110101" => sample_tone <= x"0D59D";
when "10110110" => sample_tone <= x"0D437";
when "10110111" => sample_tone <= x"0D2EF";
when "10111000" => sample_tone <= x"0D1C6";
when "10111001" => sample_tone <= x"0D0BB";
when "10111010" => sample_tone <= x"0CFD0";
when "10111011" => sample_tone <= x"0CF03";
when "10111100" => sample_tone <= x"0CE56";
when "10111101" => sample_tone <= x"0CDC8";
when "10111110" => sample_tone <= x"0CD5A";
when "10111111" => sample_tone <= x"0CD0B";
when "11000000" => sample_tone <= x"0CCDC";
when "11000001" => sample_tone <= x"0CCCC";
when "11000010" => sample_tone <= x"0CCDC";
when "11000011" => sample_tone <= x"0CD0B";
when "11000100" => sample_tone <= x"0CD5A";
when "11000101" => sample_tone <= x"0CDC9";
when "11000110" => sample_tone <= x"0CE56";
when "11000111" => sample_tone <= x"0CF04";
when "11001000" => sample_tone <= x"0CFD0";
when "11001001" => sample_tone <= x"0D0BC";
when "11001010" => sample_tone <= x"0D1C6";
when "11001011" => sample_tone <= x"0D2EF";
when "11001100" => sample_tone <= x"0D437";
when "11001101" => sample_tone <= x"0D59E";
when "11001110" => sample_tone <= x"0D722";
when "11001111" => sample_tone <= x"0D8C5";
when "11010000" => sample_tone <= x"0DA85";
when "11010001" => sample_tone <= x"0DC63";
when "11010010" => sample_tone <= x"0DE5E";
when "11010011" => sample_tone <= x"0E076";
```

```
when "11010100" => sample_tone <= x"0E2AA";
when "11010101" => sample_tone <= x"0E4FB";
when "11010110" => sample_tone <= x"0E767";
when "11010111" => sample_tone <= x"0E9EF";
when "11011000" => sample_tone <= x"0EC92";
when "11011001" => sample_tone <= x"0EF50";
when "11011010" => sample_tone <= x"0F228";
when "11011011" => sample_tone <= x"0F51A";
when "11011100" => sample_tone <= x"0F825";
when "11011101" => sample_tone <= x"0FB49";
when "11011110" => sample_tone <= x"0FE86";
when "11011111" => sample_tone <= x"101DA";
when "11100000" => sample_tone <= x"10546";
when "11100001" => sample_tone <= x"108C8";
when "11100010" => sample_tone <= x"10C61";
when "11100011" => sample_tone <= x"11010";
when "11100100" => sample_tone <= x"113D4";
when "11100101" => sample_tone <= x"117AD";
when "11100110" => sample_tone <= x"11B99";
when "11100111" => sample_tone <= x"11F99";
when "11101000" => sample_tone <= x"123AC";
when "11101001" => sample_tone <= x"127D1";
when "11101010" => sample_tone <= x"12C08";
when "11101011" => sample_tone <= x"1304F";
when "11101100" => sample_tone <= x"134A7";
when "11101101" => sample_tone <= x"1390F";
when "11101110" => sample_tone <= x"13D85";
when "11101111" => sample_tone <= x"14209";
when "11110000" => sample_tone <= x"1469B";
when "11110001" => sample_tone <= x"14B3A";
when "11110010" => sample_tone <= x"14FE4";
when "11110011" => sample_tone <= x"1549B";
when "11110100" => sample_tone <= x"1595B";
when "11110101" => sample_tone <= x"15E26";
when "11110110" => sample_tone <= x"162FA";
when "11110111" => sample_tone <= x"167D6";
when "11111000" => sample_tone <= x"16CBA";
```

```
 when "11111001" => sample_tone <= x"171A5";
 when "11111010" => sample_tone <= x"17696";
 when "11111011" => sample_tone <= x"17B8C";
 when "11111100" => sample_tone <= x"18087";
 when "11111101" => sample_tone <= x"18586";
 when "11111110" => sample_tone <= x"18A88";
 when "11111111" => sample_tone <= x"18F8D";
 when "00000000" => sample_tone <= x"19493";
 when others => sample_tone <= x"00000";
 end case;

elsif sw = '1' and tone = 1 then

case sampleCounter is
when "00000001" => sample_tone <= x"0CCCC";
when "00000010" => sample_tone <= x"0CF32";
when "00000011" => sample_tone <= x"0CF22";
when "00000100" => sample_tone <= x"0CCA5";
when "00000101" => sample_tone <= x"0C7CF";
when "00000110" => sample_tone <= x"0C0C5";
when "00000111" => sample_tone <= x"0B7B6";
when "00001000" => sample_tone <= x"0ACDC";
when "00001001" => sample_tone <= x"0A07D";
when "00001010" => sample_tone <= x"092E5";
when "00001011" => sample_tone <= x"08465";
when "00001100" => sample_tone <= x"07552";
when "00001101" => sample_tone <= x"06604";
when "00001110" => sample_tone <= x"056D0";
when "00001111" => sample_tone <= x"0480A";
when "00010000" => sample_tone <= x"039FF";
when "00010001" => sample_tone <= x"02CF7";
when "00010010" => sample_tone <= x"02130";
when "00010011" => sample_tone <= x"016DD";
when "00010100" => sample_tone <= x"00E27";
when "00010101" => sample_tone <= x"0072C";
when "00010110" => sample_tone <= x"001F8";
when "00010111" => sample_tone <= x"19828";
```

```
when "00011000" => sample_tone <= x"1967D";
when "00011001" => sample_tone <= x"19677";
when "00011010" => sample_tone <= x"197F3";
when "00011011" => sample_tone <= x"00126";
when "00011100" => sample_tone <= x"00508";
when "00011101" => sample_tone <= x"009BD";
when "00011110" => sample_tone <= x"00EFD";
when "00011111" => sample_tone <= x"0147A";
when "00100000" => sample_tone <= x"019E4";
when "00100001" => sample_tone <= x"01EE8";
when "00100010" => sample_tone <= x"0233A";
when "00100011" => sample_tone <= x"0268D";
when "00100100" => sample_tone <= x"0289C";
when "00100101" => sample_tone <= x"0292C";
when "00100110" => sample_tone <= x"02807";
when "00100111" => sample_tone <= x"02507";
when "00101000" => sample_tone <= x"02010";
when "00101001" => sample_tone <= x"01914";
when "00101010" => sample_tone <= x"01013";
when "00101011" => sample_tone <= x"0051C";
when "00101100" => sample_tone <= x"191E4";
when "00101101" => sample_tone <= x"18366";
when "00101110" => sample_tone <= x"17373";
when "00101111" => sample_tone <= x"1624F";
when "00110000" => sample_tone <= x"1504A";
when "00110001" => sample_tone <= x"13DBE";
when "00110010" => sample_tone <= x"12B0B";
when "00110011" => sample_tone <= x"11896";
when "00110100" => sample_tone <= x"106CA";
when "00110101" => sample_tone <= x"0F60F";
when "00110110" => sample_tone <= x"0E6CE";
when "00110111" => sample_tone <= x"0D96D";
when "00111000" => sample_tone <= x"0CE49";
when "00111001" => sample_tone <= x"0C5B9";
when "00111010" => sample_tone <= x"0C008";
when "00111011" => sample_tone <= x"0BD78";
when "00111100" => sample_tone <= x"0BE3A";
```

```
when "00111101" => sample_tone <= x"0C273";
when "00111110" => sample_tone <= x"0CA36";
when "00111111" => sample_tone <= x"0D587";
when "01000000" => sample_tone <= x"0E45A";
when "01000001" => sample_tone <= x"0F693";
when "01000010" => sample_tone <= x"10C04";
when "01000011" => sample_tone <= x"12473";
when "01000100" => sample_tone <= x"13F96";
when "01000101" => sample_tone <= x"15D18";
when "01000110" => sample_tone <= x"17C9A";
when "01000111" => sample_tone <= x"00419";
when "01001000" => sample_tone <= x"0265A";
when "01001001" => sample_tone <= x"04950";
when "01001010" => sample_tone <= x"06C87";
when "01001011" => sample_tone <= x"08F8B";
when "01001100" => sample_tone <= x"0B1EC";
when "01001101" => sample_tone <= x"0D33D";
when "01001110" => sample_tone <= x"0F31A";
when "01001111" => sample_tone <= x"11128";
when "01010000" => sample_tone <= x"12D17";
when "01010001" => sample_tone <= x"146A4";
when "01010010" => sample_tone <= x"15D99";
when "01010011" => sample_tone <= x"171CE";
when "01010100" => sample_tone <= x"1832A";
when "01010101" => sample_tone <= x"191A4";
when "01010110" => sample_tone <= x"19D41";
when "01010111" => sample_tone <= x"1A615";
when "01011000" => sample_tone <= x"1AC3F";
when "01011001" => sample_tone <= x"1AFEE";
when "01011010" => sample_tone <= x"1B158";
when "01011011" => sample_tone <= x"1B0BE";
when "01011100" => sample_tone <= x"1AE68";
when "01011101" => sample_tone <= x"1AAA2";
when "01011110" => sample_tone <= x"1A5BD";
when "01011111" => sample_tone <= x"1A007";
when "01100000" => sample_tone <= x"199D1";
when "01100001" => sample_tone <= x"19365";
```

```
when "01100010" => sample_tone <= x"18D09";
when "01100011" => sample_tone <= x"186FB";
when "01100100" => sample_tone <= x"18171";
when "01100101" => sample_tone <= x"17C95";
when "01100110" => sample_tone <= x"17887";
when "01100111" => sample_tone <= x"1755A";
when "01101000" => sample_tone <= x"17315";
when "01101001" => sample_tone <= x"171B0";
when "01101010" => sample_tone <= x"17119";
when "01101011" => sample_tone <= x"17131";
when "01101100" => sample_tone <= x"171CC";
when "01101101" => sample_tone <= x"172B8";
when "01101110" => sample_tone <= x"173B5";
when "01101111" => sample_tone <= x"17480";
when "01110000" => sample_tone <= x"174D1";
when "01110001" => sample_tone <= x"1745A";
when "01110010" => sample_tone <= x"172CE";
when "01110011" => sample_tone <= x"16FE2";
when "01110100" => sample_tone <= x"16B4E";
when "01110101" => sample_tone <= x"164CD";
when "01110110" => sample_tone <= x"15C25";
when "01110111" => sample_tone <= x"15124";
when "01111000" => sample_tone <= x"143A2";
when "01111001" => sample_tone <= x"13386";
when "01111010" => sample_tone <= x"120C1";
when "01111011" => sample_tone <= x"10B55";
when "01111100" => sample_tone <= x"0F351";
when "01111101" => sample_tone <= x"0D8D5";
when "01111110" => sample_tone <= x"0BC0F";
when "01111111" => sample_tone <= x"09D38";
when "10000000" => sample_tone <= x"07C9A";
when "10000001" => sample_tone <= x"05A8A";
when "10000010" => sample_tone <= x"03767";
when "10000011" => sample_tone <= x"01397";
when "10000100" => sample_tone <= x"18922";
when "10000101" => sample_tone <= x"16549";
when "10000110" => sample_tone <= x"14219";
```

```
when "10000111" => sample_tone <= x"12006";
when "10001000" => sample_tone <= x"0FF84";
when "10001001" => sample_tone <= x"0E0FE";
when "10001010" => sample_tone <= x"0C4DB";
when "10001011" => sample_tone <= x"0AB77";
when "10001100" => sample_tone <= x"09524";
when "10001101" => sample_tone <= x"08225";
when "10001110" => sample_tone <= x"072B2";
when "10001111" => sample_tone <= x"066F1";
when "10010000" => sample_tone <= x"05EF9";
when "10010001" => sample_tone <= x"05AD1";
when "10010010" => sample_tone <= x"05A6E";
when "10010011" => sample_tone <= x"05DB6";
when "10010100" => sample_tone <= x"0647F";
when "10010101" => sample_tone <= x"06E91";
when "10010110" => sample_tone <= x"07BA5";
when "10010111" => sample_tone <= x"08B6A";
when "10011000" => sample_tone <= x"09D85";
when "10011001" => sample_tone <= x"0B193";
when "10011010" => sample_tone <= x"0C72B";
when "10011011" => sample_tone <= x"0DDE1";
when "10011100" => sample_tone <= x"0F548";
when "10011101" => sample_tone <= x"10CF6";
when "10011110" => sample_tone <= x"12482";
when "10011111" => sample_tone <= x"13B8A";
when "10100000" => sample_tone <= x"151B5";
when "10100001" => sample_tone <= x"166B3";
when "10100010" => sample_tone <= x"17A3F";
when "10100011" => sample_tone <= x"18C20";
when "10100100" => sample_tone <= x"00292";
when "10100101" => sample_tone <= x"010AC";
when "10100110" => sample_tone <= x"01CC6";
when "10100111" => sample_tone <= x"026E1";
when "10101000" => sample_tone <= x"02F0A";
when "10101001" => sample_tone <= x"0355E";
when "10101010" => sample_tone <= x"03A04";
when "10101011" => sample_tone <= x"03D2F";
```

```
when "10101100" => sample_tone <= x"03F1B";
when "10101101" => sample_tone <= x"0400B";
when "10101110" => sample_tone <= x"04049";
when "10101111" => sample_tone <= x"04020";
when "10110000" => sample_tone <= x"03FDF";
when "10110001" => sample_tone <= x"03FD2";
when "10110010" => sample_tone <= x"04042";
when "10110011" => sample_tone <= x"04174";
when "10110100" => sample_tone <= x"043A5";
when "10110101" => sample_tone <= x"0470B";
when "10110110" => sample_tone <= x"04BCF";
when "10110111" => sample_tone <= x"05210";
when "10111000" => sample_tone <= x"059E2";
when "10111001" => sample_tone <= x"06349";
when "10111010" => sample_tone <= x"06E3E";
when "10111011" => sample_tone <= x"07AAC";
when "10111100" => sample_tone <= x"08871";
when "10111101" => sample_tone <= x"09760";
when "10111110" => sample_tone <= x"0A73F";
when "10111111" => sample_tone <= x"0B7CD";
when "11000000" => sample_tone <= x"0C8BE";
when "11000001" => sample_tone <= x"0D9C2";
when "11000010" => sample_tone <= x"0EA83";
when "11000011" => sample_tone <= x"0FAAB";
when "11000100" => sample_tone <= x"109E2";
when "11000101" => sample_tone <= x"117D3";
when "11000110" => sample_tone <= x"1242C";
when "11000111" => sample_tone <= x"12EA5";
when "11001000" => sample_tone <= x"136F9";
when "11001001" => sample_tone <= x"13CF3";
when "11001010" => sample_tone <= x"14065";
when "11001011" => sample_tone <= x"14131";
when "11001100" => sample_tone <= x"13F45";
when "11001101" => sample_tone <= x"13A9E";
when "11001110" => sample_tone <= x"1334A";
when "11001111" => sample_tone <= x"12964";
when "11010000" => sample_tone <= x"11D14";
```

```
when "11010001" => sample_tone <= x"10E93";
when "11010010" => sample_tone <= x"0FE25";
when "11010011" => sample_tone <= x"0EC19";
when "11010100" => sample_tone <= x"0D8C9";
when "11010101" => sample_tone <= x"0C495";
when "11010110" => sample_tone <= x"0AFE6";
when "11010111" => sample_tone <= x"09B25";
when "11011000" => sample_tone <= x"086BE";
when "11011001" => sample_tone <= x"0731B";
when "11011010" => sample_tone <= x"060A5";
when "11011011" => sample_tone <= x"04FBC";
when "11011100" => sample_tone <= x"040BB";
when "11011101" => sample_tone <= x"033F0";
when "11011110" => sample_tone <= x"029A1";
when "11011111" => sample_tone <= x"02204";
when "11100000" => sample_tone <= x"01D41";
when "11100001" => sample_tone <= x"01B73";
when "11100010" => sample_tone <= x"01CA3";
when "11100011" => sample_tone <= x"020C9";
when "11100100" => sample_tone <= x"027D0";
when "11100101" => sample_tone <= x"03192";
when "11100110" => sample_tone <= x"03DD9";
when "11100111" => sample_tone <= x"04C66";
when "11101000" => sample_tone <= x"05CE8";
when "11101001" => sample_tone <= x"06F09";
when "11101010" => sample_tone <= x"08269";
when "11101011" => sample_tone <= x"096A0";
when "11101100" => sample_tone <= x"0AB46";
when "11101101" => sample_tone <= x"0BFED";
when "11101110" => sample_tone <= x"0D42B";
when "11101111" => sample_tone <= x"0E799";
when "11110000" => sample_tone <= x"0F9D5";
when "11110001" => sample_tone <= x"10A82";
when "11110010" => sample_tone <= x"11951";
when "11110011" => sample_tone <= x"125FA";
when "11110100" => sample_tone <= x"13044";
when "11110101" => sample_tone <= x"13803";
```

```
   when "11110110" => sample_tone <= x"13D19";
   when "11110111" => sample_tone <= x"13F76";
   when "11111000" => sample_tone <= x"13F1B";
   when "11111001" => sample_tone <= x"13C17";
   when "11111010" => sample_tone <= x"13686";
   when "11111011" => sample_tone <= x"12E92";
   when "11111100" => sample_tone <= x"12470";
   when "11111101" => sample_tone <= x"11862";
   when "11111110" => sample_tone <= x"10AB0";
   when "11111111" => sample_tone <= x"0FBAB";
   when "00000000" => sample_tone <= x"0EBA6";
   when others => sample_tone <= x"0DAF8";

   end case;

elsif sw = '1' and tone = 2 then
    case sampleCounter is
when "00000001" => sample_tone <= x"13332";
    when "00000010" => sample_tone <= x"13D86";
    when "00000011" => sample_tone <= x"14864";
    when "00000100" => sample_tone <= x"153C7";
    when "00000101" => sample_tone <= x"15FA8";
    when "00000110" => sample_tone <= x"16C02";
    when "00000111" => sample_tone <= x"178CE";
    when "00001000" => sample_tone <= x"18602";
    when "00001001" => sample_tone <= x"19398";
    when "00001010" => sample_tone <= x"007ED";
    when "00001011" => sample_tone <= x"01629";
    when "00001100" => sample_tone <= x"024A8";
    when "00001101" => sample_tone <= x"03361";
    when "00001110" => sample_tone <= x"04247";
    when "00001111" => sample_tone <= x"05150";
    when "00010000" => sample_tone <= x"0606F";
    when "00010001" => sample_tone <= x"06F98";
    when "00010010" => sample_tone <= x"07EBE";
    when "00010011" => sample_tone <= x"08DD6";
    when "00010100" => sample_tone <= x"09CD2";
```

```
when "00010101" => sample_tone <= x"0ABA5";
when "00010110" => sample_tone <= x"0BA43";
when "00010111" => sample_tone <= x"0C89E";
when "00011000" => sample_tone <= x"0D6AB";
when "00011001" => sample_tone <= x"0E45C";
when "00011010" => sample_tone <= x"0F1A6";
when "00011011" => sample_tone <= x"0FE7C";
when "00011100" => sample_tone <= x"10AD3";
when "00011101" => sample_tone <= x"116A0";
when "00011110" => sample_tone <= x"121D8";
when "00011111" => sample_tone <= x"12C70";
when "00100000" => sample_tone <= x"13660";
when "00100001" => sample_tone <= x"13F9F";
when "00100010" => sample_tone <= x"14823";
when "00100011" => sample_tone <= x"14FE6";
when "00100100" => sample_tone <= x"156E1";
when "00100101" => sample_tone <= x"15D0D";
when "00100110" => sample_tone <= x"16265";
when "00100111" => sample_tone <= x"166E6";
when "00101000" => sample_tone <= x"16A8B";
when "00101001" => sample_tone <= x"16D52";
when "00101010" => sample_tone <= x"16F38";
when "00101011" => sample_tone <= x"1703F";
when "00101100" => sample_tone <= x"17064";
when "00101101" => sample_tone <= x"16FAA";
when "00101110" => sample_tone <= x"16E12";
when "00101111" => sample_tone <= x"16B9F";
when "00110000" => sample_tone <= x"16855";
when "00110001" => sample_tone <= x"16438";
when "00110010" => sample_tone <= x"15F4E";
when "00110011" => sample_tone <= x"1599E";
when "00110100" => sample_tone <= x"1532E";
when "00110101" => sample_tone <= x"14C06";
when "00110110" => sample_tone <= x"1442F";
when "00110111" => sample_tone <= x"13BB2";
when "00111000" => sample_tone <= x"1329A";
when "00111001" => sample_tone <= x"128F0";
```

```
when "00111010" => sample_tone <= x"11EC0";
when "00111011" => sample_tone <= x"11416";
when "00111100" => sample_tone <= x"108FD";
when "00111101" => sample_tone <= x"0FD83";
when "00111110" => sample_tone <= x"0F1B4";
when "00111111" => sample_tone <= x"0E59C";
when "01000000" => sample_tone <= x"0D94A";
when "01000001" => sample_tone <= x"0CCCC";
when "01000010" => sample_tone <= x"0C02D";
when "01000011" => sample_tone <= x"0B37C";
when "01000100" => sample_tone <= x"0A6C7";
when "01000101" => sample_tone <= x"09A1B";
when "01000110" => sample_tone <= x"08D85";
when "01000111" => sample_tone <= x"08112";
when "01001000" => sample_tone <= x"074CF";
when "01001001" => sample_tone <= x"068C8";
when "01001010" => sample_tone <= x"05D0A";
when "01001011" => sample_tone <= x"0519F";
when "01001100" => sample_tone <= x"04692";
when "01001101" => sample_tone <= x"03BEE";
when "01001110" => sample_tone <= x"031BD";
when "01001111" => sample_tone <= x"02808";
when "01010000" => sample_tone <= x"01ED7";
when "01010001" => sample_tone <= x"01632";
when "01010010" => sample_tone <= x"00E1F";
when "01010011" => sample_tone <= x"006A5";
when "01010100" => sample_tone <= x"19963";
when "01010101" => sample_tone <= x"1932A";
when "01010110" => sample_tone <= x"18D97";
when "01010111" => sample_tone <= x"188AC";
when "01011000" => sample_tone <= x"1846C";
when "01011001" => sample_tone <= x"180D8";
when "01011010" => sample_tone <= x"17DEF";
when "01011011" => sample_tone <= x"17BB0";
when "01011100" => sample_tone <= x"17A1A";
when "01011101" => sample_tone <= x"1792B";
when "01011110" => sample_tone <= x"178DE";
```

```
when "01011111" => sample_tone <= x"17930";
when "01100000" => sample_tone <= x"17A1B";
when "01100001" => sample_tone <= x"17B9B";
when "01100010" => sample_tone <= x"17DA7";
when "01100011" => sample_tone <= x"1803A";
when "01100100" => sample_tone <= x"1834A";
when "01100101" => sample_tone <= x"186D1";
when "01100110" => sample_tone <= x"18AC5";
when "01100111" => sample_tone <= x"18F1C";
when "01101000" => sample_tone <= x"193CC";
when "01101001" => sample_tone <= x"198CC";
when "01101010" => sample_tone <= x"00477";
when "01101011" => sample_tone <= x"009F4";
when "01101100" => sample_tone <= x"00FA0";
when "01101101" => sample_tone <= x"01570";
when "01101110" => sample_tone <= x"01B56";
when "01101111" => sample_tone <= x"0214A";
when "01110000" => sample_tone <= x"0273E";
when "01110001" => sample_tone <= x"02D27";
when "01110010" => sample_tone <= x"032FA";
when "01110011" => sample_tone <= x"038AD";
when "01110100" => sample_tone <= x"03E34";
when "01110101" => sample_tone <= x"04385";
when "01110110" => sample_tone <= x"04896";
when "01110111" => sample_tone <= x"04D5D";
when "01111000" => sample_tone <= x"051D1";
when "01111001" => sample_tone <= x"055E9";
when "01111010" => sample_tone <= x"0599C";
when "01111011" => sample_tone <= x"05CE4";
when "01111100" => sample_tone <= x"05FBA";
when "01111101" => sample_tone <= x"06216";
when "01111110" => sample_tone <= x"063F3";
when "01111111" => sample_tone <= x"0654D";
when "10000000" => sample_tone <= x"0661F";
when "10000001" => sample_tone <= x"06666";
when "10000010" => sample_tone <= x"0661E";
when "10000011" => sample_tone <= x"06547";
```

```
when "10000100" => sample_tone <= x"063DE";
when "10000101" => sample_tone <= x"061E4";
when "10000110" => sample_tone <= x"05F59";
when "10000111" => sample_tone <= x"05C3D";
when "10001000" => sample_tone <= x"05894";
when "10001001" => sample_tone <= x"0545F";
when "10001010" => sample_tone <= x"04FA2";
when "10001011" => sample_tone <= x"04A61";
when "10001100" => sample_tone <= x"044A0";
when "10001101" => sample_tone <= x"03E66";
when "10001110" => sample_tone <= x"037B7";
when "10001111" => sample_tone <= x"0309B";
when "10010000" => sample_tone <= x"02919";
when "10010001" => sample_tone <= x"02137";
when "10010010" => sample_tone <= x"018FF";
when "10010011" => sample_tone <= x"01078";
when "10010100" => sample_tone <= x"007AB";
when "10010101" => sample_tone <= x"1983B";
when "10010110" => sample_tone <= x"18EFE";
when "10010111" => sample_tone <= x"18597";
when "10011000" => sample_tone <= x"17C0F";
when "10011001" => sample_tone <= x"17271";
when "10011010" => sample_tone <= x"168C5";
when "10011011" => sample_tone <= x"15F16";
when "10011100" => sample_tone <= x"1556D";
when "10011101" => sample_tone <= x"14BD5";
when "10011110" => sample_tone <= x"14255";
when "10011111" => sample_tone <= x"138F7";
when "10100000" => sample_tone <= x"12FC4";
when "10100001" => sample_tone <= x"126C5";
when "10100010" => sample_tone <= x"11E01";
when "10100011" => sample_tone <= x"11581";
when "10100100" => sample_tone <= x"10D4C";
when "10100101" => sample_tone <= x"10568";
when "10100110" => sample_tone <= x"0FDDB";
when "10100111" => sample_tone <= x"0F6AC";
when "10101000" => sample_tone <= x"0EFE0";
```

```
when "10101001" => sample_tone <= x"0E97B";
when "10101010" => sample_tone <= x"0E382";
when "10101011" => sample_tone <= x"0DDF7";
when "10101100" => sample_tone <= x"0D8DD";
when "10101101" => sample_tone <= x"0D437";
when "10101110" => sample_tone <= x"0D005";
when "10101111" => sample_tone <= x"0CC49";
when "10110000" => sample_tone <= x"0C902";
when "10110001" => sample_tone <= x"0C630";
when "10110010" => sample_tone <= x"0C3D2";
when "10110011" => sample_tone <= x"0C1E6";
when "10110100" => sample_tone <= x"0C069";
when "10110101" => sample_tone <= x"0BF59";
when "10110110" => sample_tone <= x"0BEB2";
when "10110111" => sample_tone <= x"0BE70";
when "10111000" => sample_tone <= x"0BE8E";
when "10111001" => sample_tone <= x"0BF07";
when "10111010" => sample_tone <= x"0BFD6";
when "10111011" => sample_tone <= x"0C0F5";
when "10111100" => sample_tone <= x"0C25E";
when "10111101" => sample_tone <= x"0C409";
when "10111110" => sample_tone <= x"0C5F1";
when "10111111" => sample_tone <= x"0C80E";
when "11000000" => sample_tone <= x"0CA5A";
when "11000001" => sample_tone <= x"0CCCC";
when "11000010" => sample_tone <= x"0CF5E";
when "11000011" => sample_tone <= x"0D208";
when "11000100" => sample_tone <= x"0D4C4";
when "11000101" => sample_tone <= x"0D788";
when "11000110" => sample_tone <= x"0DA4F";
when "11000111" => sample_tone <= x"0DD12";
when "11001000" => sample_tone <= x"0DFCA";
when "11001001" => sample_tone <= x"0E270";
when "11001010" => sample_tone <= x"0E4FE";
when "11001011" => sample_tone <= x"0E76F";
when "11001100" => sample_tone <= x"0E9BC";
when "11001101" => sample_tone <= x"0EBE2";
```

```
when "11001110" => sample_tone <= x"0EDDB";
when "11001111" => sample_tone <= x"0EFA4";
when "11010000" => sample_tone <= x"0F138";
when "11010001" => sample_tone <= x"0F296";
when "11010010" => sample_tone <= x"0F3BA";
when "11010011" => sample_tone <= x"0F4A3";
when "11010100" => sample_tone <= x"0F54F";
when "11010101" => sample_tone <= x"0F5BE";
when "11010110" => sample_tone <= x"0F5F1";
when "11010111" => sample_tone <= x"0F5E7";
when "11011000" => sample_tone <= x"0F5A2";
when "11011001" => sample_tone <= x"0F523";
when "11011010" => sample_tone <= x"0F46E";
when "11011011" => sample_tone <= x"0F386";
when "11011100" => sample_tone <= x"0F26D";
when "11011101" => sample_tone <= x"0F129";
when "11011110" => sample_tone <= x"0EFBE";
when "11011111" => sample_tone <= x"0EE31";
when "11100000" => sample_tone <= x"0EC88";
when "11100001" => sample_tone <= x"0EACA";
when "11100010" => sample_tone <= x"0E8FC";
when "11100011" => sample_tone <= x"0E727";
when "11100100" => sample_tone <= x"0E551";
when "11100101" => sample_tone <= x"0E383";
when "11100110" => sample_tone <= x"0E1C3";
when "11100111" => sample_tone <= x"0E01B";
when "11101000" => sample_tone <= x"0DE91";
when "11101001" => sample_tone <= x"0DD30";
when "11101010" => sample_tone <= x"0DBFF";
when "11101011" => sample_tone <= x"0DB06";
when "11101100" => sample_tone <= x"0DA4E";
when "11101101" => sample_tone <= x"0D9E0";
when "11101110" => sample_tone <= x"0D9C3";
when "11101111" => sample_tone <= x"0D9FF";
when "11110000" => sample_tone <= x"0DA9C";
when "11110001" => sample_tone <= x"0DBA1";
when "11110010" => sample_tone <= x"0DD15";
```

```
     when "11110011" => sample_tone <= x"0DEFF";
     when "11110100" => sample_tone <= x"0E165";
     when "11110101" => sample_tone <= x"0E44C";
     when "11110110" => sample_tone <= x"0E7B9";
     when "11110111" => sample_tone <= x"0EBB1";
     when "11111000" => sample_tone <= x"0F038";
     when "11111001" => sample_tone <= x"0F551";
     when "11111010" => sample_tone <= x"0FAFE";
     when "11111011" => sample_tone <= x"10142";
     when "11111100" => sample_tone <= x"1081D";
     when "11111101" => sample_tone <= x"10F8F";
     when "11111110" => sample_tone <= x"11799";
     when "11111111" => sample_tone <= x"12039";
     when "00000000" => sample_tone <= x"1296E";
     when others => sample_tone <= x"13334";
    end case;

else
     case sampleCounter is
    when "00000001" => sample_tone <= x"00000";
     when "00000010" => sample_tone <= x"00000";
     when "00000011" => sample_tone <= x"00000";
     when "00000100" => sample_tone <= x"00000";
     when "00000101" => sample_tone <= x"00000";
     when "00000110" => sample_tone <= x"00000";
     when "00000111" => sample_tone <= x"00000";
     when "00001000" => sample_tone <= x"00000";
     when "00001001" => sample_tone <= x"00000";
     when "00001010" => sample_tone <= x"00000";
     when "00001011" => sample_tone <= x"00000";
     when "00001100" => sample_tone <= x"00000";
     when "00001101" => sample_tone <= x"00000";
     when "00001110" => sample_tone <= x"00000";
     when "00001111" => sample_tone <= x"00000";
     when "00010000" => sample_tone <= x"00000";
     when "00010001" => sample_tone <= x"00000";
     when "00010010" => sample_tone <= x"00000";
```

```
when "00010011" => sample_tone <= x"00000";
when "00010100" => sample_tone <= x"00000";
when "00010101" => sample_tone <= x"00000";
when "00010110" => sample_tone <= x"00000";
when "00010111" => sample_tone <= x"00000";
when "00011000" => sample_tone <= x"00000";
when "00011001" => sample_tone <= x"00000";
when "00011010" => sample_tone <= x"00000";
when "00011011" => sample_tone <= x"00000";
when "00011100" => sample_tone <= x"00000";
when "00011101" => sample_tone <= x"00000";
when "00011110" => sample_tone <= x"00000";
when "00011111" => sample_tone <= x"00000";
when "00100000" => sample_tone <= x"00000";
when "00100001" => sample_tone <= x"00000";
when "00100010" => sample_tone <= x"00000";
when "00100011" => sample_tone <= x"00000";
when "00100100" => sample_tone <= x"00000";
when "00100101" => sample_tone <= x"00000";
when "00100110" => sample_tone <= x"00000";
when "00100111" => sample_tone <= x"00000";
when "00101000" => sample_tone <= x"00000";
when "00101001" => sample_tone <= x"00000";
when "00101010" => sample_tone <= x"00000";
when "00101011" => sample_tone <= x"00000";
when "00101100" => sample_tone <= x"00000";
when "00101101" => sample_tone <= x"00000";
when "00101110" => sample_tone <= x"00000";
when "00101111" => sample_tone <= x"00000";
when "00110000" => sample_tone <= x"00000";
when "00110001" => sample_tone <= x"00000";
when "00110010" => sample_tone <= x"00000";
when "00110011" => sample_tone <= x"00000";
when "00110100" => sample_tone <= x"00000";
when "00110101" => sample_tone <= x"00000";
when "00110110" => sample_tone <= x"00000";
when "00110111" => sample_tone <= x"00000";
```

```
when "00111000" => sample_tone <= x"00000";
when "00111001" => sample_tone <= x"00000";
when "00111010" => sample_tone <= x"00000";
when "00111011" => sample_tone <= x"00000";
when "00111100" => sample_tone <= x"00000";
when "00111101" => sample_tone <= x"00000";
when "00111110" => sample_tone <= x"00000";
when "00111111" => sample_tone <= x"00000";
when "01000000" => sample_tone <= x"00000";
when "01000001" => sample_tone <= x"00000";
when "01000010" => sample_tone <= x"00000";
when "01000011" => sample_tone <= x"00000";
when "01000100" => sample_tone <= x"00000";
when "01000101" => sample_tone <= x"00000";
when "01000110" => sample_tone <= x"00000";
when "01000111" => sample_tone <= x"00000";
when "01001000" => sample_tone <= x"00000";
when "01001001" => sample_tone <= x"00000";
when "01001010" => sample_tone <= x"00000";
when "01001011" => sample_tone <= x"00000";
when "01001100" => sample_tone <= x"00000";
when "01001101" => sample_tone <= x"00000";
when "01001110" => sample_tone <= x"00000";
when "01001111" => sample_tone <= x"00000";
when "01010000" => sample_tone <= x"00000";
when "01010001" => sample_tone <= x"00000";
when "01010010" => sample_tone <= x"00000";
when "01010011" => sample_tone <= x"00000";
when "01010100" => sample_tone <= x"00000";
when "01010101" => sample_tone <= x"00000";
when "01010110" => sample_tone <= x"00000";
when "01010111" => sample_tone <= x"00000";
when "01011000" => sample_tone <= x"00000";
when "01011001" => sample_tone <= x"00000";
when "01011010" => sample_tone <= x"00000";
when "01011011" => sample_tone <= x"00000";
when "01011100" => sample_tone <= x"00000";
```

```
when "01011101" => sample_tone <= x"00000";
when "01011110" => sample_tone <= x"00000";
when "01011111" => sample_tone <= x"00000";
when "01100000" => sample_tone <= x"00000";
when "01100001" => sample_tone <= x"00000";
when "01100010" => sample_tone <= x"00000";
when "01100011" => sample_tone <= x"00000";
when "01100100" => sample_tone <= x"00000";
when "01100101" => sample_tone <= x"00000";
when "01100110" => sample_tone <= x"00000";
when "01100111" => sample_tone <= x"00000";
when "01101000" => sample_tone <= x"00000";
when "01101001" => sample_tone <= x"00000";
when "01101010" => sample_tone <= x"00000";
when "01101011" => sample_tone <= x"00000";
when "01101100" => sample_tone <= x"00000";
when "01101101" => sample_tone <= x"00000";
when "01101110" => sample_tone <= x"00000";
when "01101111" => sample_tone <= x"00000";
when "01110000" => sample_tone <= x"00000";
when "01110001" => sample_tone <= x"00000";
when "01110010" => sample_tone <= x"00000";
when "01110011" => sample_tone <= x"00000";
when "01110100" => sample_tone <= x"00000";
when "01110101" => sample_tone <= x"00000";
when "01110110" => sample_tone <= x"00000";
when "01110111" => sample_tone <= x"00000";
when "01111000" => sample_tone <= x"00000";
when "01111001" => sample_tone <= x"00000";
when "01111010" => sample_tone <= x"00000";
when "01111011" => sample_tone <= x"00000";
when "01111100" => sample_tone <= x"00000";
when "01111101" => sample_tone <= x"00000";
when "01111110" => sample_tone <= x"00000";
when "01111111" => sample_tone <= x"00000";
when "10000000" => sample_tone <= x"00000";
when "10000001" => sample_tone <= x"00000";
```

```
when "10000010" => sample_tone <= x"00000";
when "10000011" => sample_tone <= x"00000";
when "10000100" => sample_tone <= x"00000";
when "10000101" => sample_tone <= x"00000";
when "10000110" => sample_tone <= x"00000";
when "10000111" => sample_tone <= x"00000";
when "10001000" => sample_tone <= x"00000";
when "10001001" => sample_tone <= x"00000";
when "10001010" => sample_tone <= x"00000";
when "10001011" => sample_tone <= x"00000";
when "10001100" => sample_tone <= x"00000";
when "10001101" => sample_tone <= x"00000";
when "10001110" => sample_tone <= x"00000";
when "10001111" => sample_tone <= x"00000";
when "10010000" => sample_tone <= x"00000";
when "10010001" => sample_tone <= x"00000";
when "10010010" => sample_tone <= x"00000";
when "10010011" => sample_tone <= x"00000";
when "10010100" => sample_tone <= x"00000";
when "10010101" => sample_tone <= x"00000";
when "10010110" => sample_tone <= x"00000";
when "10010111" => sample_tone <= x"00000";
when "10011000" => sample_tone <= x"00000";
when "10011001" => sample_tone <= x"00000";
when "10011010" => sample_tone <= x"00000";
when "10011011" => sample_tone <= x"00000";
when "10011100" => sample_tone <= x"00000";
when "10011101" => sample_tone <= x"00000";
when "10011110" => sample_tone <= x"00000";
when "10011111" => sample_tone <= x"00000";
when "10100000" => sample_tone <= x"00000";
when "10100001" => sample_tone <= x"00000";
when "10100010" => sample_tone <= x"00000";
when "10100011" => sample_tone <= x"00000";
when "10100100" => sample_tone <= x"00000";
when "10100101" => sample_tone <= x"00000";
when "10100110" => sample_tone <= x"00000";
```

```
when "10100111" => sample_tone <= x"00000";
when "10101000" => sample_tone <= x"00000";
when "10101001" => sample_tone <= x"00000";
when "10101010" => sample_tone <= x"00000";
when "10101011" => sample_tone <= x"00000";
when "10101100" => sample_tone <= x"00000";
when "10101101" => sample_tone <= x"00000";
when "10101110" => sample_tone <= x"00000";
when "10101111" => sample_tone <= x"00000";
when "10110000" => sample_tone <= x"00000";
when "10110001" => sample_tone <= x"00000";
when "10110010" => sample_tone <= x"00000";
when "10110011" => sample_tone <= x"00000";
when "10110100" => sample_tone <= x"00000";
when "10110101" => sample_tone <= x"00000";
when "10110110" => sample_tone <= x"00000";
when "10110111" => sample_tone <= x"00000";
when "10111000" => sample_tone <= x"00000";
when "10111001" => sample_tone <= x"00000";
when "10111010" => sample_tone <= x"00000";
when "10111011" => sample_tone <= x"00000";
when "10111100" => sample_tone <= x"00000";
when "10111101" => sample_tone <= x"00000";
when "10111110" => sample_tone <= x"00000";
when "10111111" => sample_tone <= x"00000";
when "11000000" => sample_tone <= x"00000";
when "11000001" => sample_tone <= x"00000";
when "11000010" => sample_tone <= x"00000";
when "11000011" => sample_tone <= x"00000";
when "11000100" => sample_tone <= x"00000";
when "11000101" => sample_tone <= x"00000";
when "11000110" => sample_tone <= x"00000";
when "11000111" => sample_tone <= x"00000";
when "11001000" => sample_tone <= x"00000";
when "11001001" => sample_tone <= x"00000";
when "11001010" => sample_tone <= x"00000";
when "11001011" => sample_tone <= x"00000";
```

```
when "11001100" => sample_tone <= x"00000";
when "11001101" => sample_tone <= x"00000";
when "11001110" => sample_tone <= x"00000";
when "11001111" => sample_tone <= x"00000";
when "11010000" => sample_tone <= x"00000";
when "11010001" => sample_tone <= x"00000";
when "11010010" => sample_tone <= x"00000";
when "11010011" => sample_tone <= x"00000";
when "11010100" => sample_tone <= x"00000";
when "11010101" => sample_tone <= x"00000";
when "11010110" => sample_tone <= x"00000";
when "11010111" => sample_tone <= x"00000";
when "11011000" => sample_tone <= x"00000";
when "11011001" => sample_tone <= x"00000";
when "11011010" => sample_tone <= x"00000";
when "11011011" => sample_tone <= x"00000";
when "11011100" => sample_tone <= x"00000";
when "11011101" => sample_tone <= x"00000";
when "11011110" => sample_tone <= x"00000";
when "11011111" => sample_tone <= x"00000";
when "11100000" => sample_tone <= x"00000";
when "11100001" => sample_tone <= x"00000";
when "11100010" => sample_tone <= x"00000";
when "11100011" => sample_tone <= x"00000";
when "11100100" => sample_tone <= x"00000";
when "11100101" => sample_tone <= x"00000";
when "11100110" => sample_tone <= x"00000";
when "11100111" => sample_tone <= x"00000";
when "11101000" => sample_tone <= x"00000";
when "11101001" => sample_tone <= x"00000";
when "11101010" => sample_tone <= x"00000";
when "11101011" => sample_tone <= x"00000";
when "11101100" => sample_tone <= x"00000";
when "11101101" => sample_tone <= x"00000";
when "11101110" => sample_tone <= x"00000";
when "11101111" => sample_tone <= x"00000";
when "11110000" => sample_tone <= x"00000";
```

```
    when "11110001" => sample_tone <= x"00000";
    when "11110010" => sample_tone <= x"00000";
    when "11110011" => sample_tone <= x"00000";
    when "11110100" => sample_tone <= x"00000";
    when "11110101" => sample_tone <= x"00000";
    when "11110110" => sample_tone <= x"00000";
    when "11110111" => sample_tone <= x"00000";
    when "11111000" => sample_tone <= x"00000";
    when "11111001" => sample_tone <= x"00000";
    when "11111010" => sample_tone <= x"00000";
    when "11111011" => sample_tone <= x"00000";
    when "11111100" => sample_tone <= x"00000";
    when "11111101" => sample_tone <= x"00000";
    when "11111110" => sample_tone <= x"00000";
    when "11111111" => sample_tone <= x"00000";
    when "00000000" => sample_tone <= x"00000";
    when others => sample_tone <= x"00000";
    end case;
   end if;

end process;

process (volume, sample_tone, velo) begin

    if velo > 0 then
        value <= to_integer(signed(sample_tone)) * volume;
        sample <= std_logic_vector(to_signed(value,20));
    else
        sample <= "00000000000000000000";
    end if;
end process;

end Behavioral;
```

10.2. Sinthesizer: *Constraints*

L'arxiu de *constriants* defineix els ports actius de l'FPGA i els enllaça amb les senyals d'entrada i sortida del programa Synthesizer.

10.2.1 const_synth3v3_1.*vhd*

```
set_property IOSTANDARD LVCMOS33 [get_ports sound_on_off]
set_property IOSTANDARD LVCMOS33 [get_ports midi_in]
set_property IOSTANDARD LVCMOS33 [get_ports spdif_out]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property PACKAGE_PIN W5 [get_ports clk]

set_property PACKAGE_PIN R2 [get_ports sound_on_off]
set_property PACKAGE_PIN B16 [get_ports midi_in]
set_property PACKAGE_PIN L1 [get_ports spdif_out]

set_property IOSTANDARD LVCMOS33 [get_ports vol_down]
set_property IOSTANDARD LVCMOS33 [get_ports vol_up]
set_property PACKAGE_PIN U17 [get_ports vol_down]
set_property PACKAGE_PIN T18 [get_ports vol_up]

set_property IOSTANDARD LVCMOS33 [get_ports tone_down]
set_property IOSTANDARD LVCMOS33 [get_ports tone_up]
set_property PACKAGE_PIN T17 [get_ports tone_down]
set_property PACKAGE_PIN W19 [get_ports tone_up]
```